Rapport du projet d'IFB : Jeu de Belote Coinchée en C dans la console

Table des matières

Introduction

Dans le cadre de l'UV IFB nous avons dû réaliser un jeu de Belote Coinchée en [1]C. Le cahier des charges nous impose de programmer uniquement en C et de faire un programme qui s'exécute dans la console.

Dans le cadre de l'UV IFB nous avons dû réaliser un jeu de Belote Coinchée en C. Le cahier des charges nous impose de programmer uniquement en C et de faire un programme qui s'exécute dans la console.

Etant donné que nous avons appris en cour à développer avec le logiciel Code :: Block, nous avons décidé que nous réaliserons ce projet sur Code :: Block car ce logiciel bien qui soit assez ancien et plus mis a jour depuis plusieurs années reste beaucoup plus facile à prendre en main que d'autres ide plus complexe mais plus puissant comme Visual Studio code par exemple. A postériori, nous somme content d'avoir fait ce choix et nous avons pu profiter de certaine fonctionnalité proposée par ce logiciel.

Afin de produire un code de la meilleure qualité possible nous nous sommes mis d'accord sur certaine bonne manière de programmer avant même de commencer le projet. Ainsi nous avons décidé d'adopter un nomenclature commune pour les variables et les fonctions (nomDeLaVariable, nomDeLaFonction) et une autres pour les struct et les enum(NomDuStruct). De plus afin d'améliorer la lisibilité et rendre le code facile a améliorer et à mettre à jour, nous avons décidé que tout ce qui serai répétée au moins deux fois dans le programme devrait être mis dans une fonction et que les fonction devait être placée dans différent ficher selon leur utilité.

Toujours dans l'objectif de produire un code bonne qualité, nous avons documenter et ajouté des commentaires afin d'expliciter un maximum de chose. Ainsi toutes les fonctions, toutes les structures et tous les fichiers du projet sont accompagné d'une description placée dans un bloc de commentaire respectant la norme du logiciel Doxigen. Grâce à cela nous avons, une fois le projet terminé put exporter à l'aide de Doxigen toute la documentation présente dans notre projet. Ce qui a permis de créer le fichier documentation.pdf qui regroupe toutes les informations que nous avons placer dans le code sur les différentes fonctions ou structures. Grâce à ce document il est alors beaucoup plus simple pour une personne extérieure au projet de comprendre comment s'utilise nos fonctions ou nos structures.

Sources utilisées

https://www.belote.com/regles-et-variantes/regle-belote-coinche/

http://www.ffbelote.org/regles-coinche/#7

https://mon.gameduell.fr/gd/s03.do?gametype=bel&top

https://openclassrooms.com/fr/courses/19980-apprenez-a-programmer-en-c

https://openclassrooms.com/fr/courses/2342361-gerez-votre-code-avec-git-et-github

https://www.youtube.com/watch?v=x7IlDycK04M

https://stackoverflow.com/

Organisation générale du projet

Afin de réaliser ce projet et sachant qu'à cause du confinement il serait très difficile pour les membres du projet de se retrouver en présentiel pour se coordonner sur le projet, nous avons utilisé la plateforme discord afin de communiquer sur les problèmes que nous avons rencontrés et sur les fonctions à réaliser. Pour stoker et versionner notre code, nous avons utilisé la plateforme GitHub qui nous permet de travailler simultanément sur le projet et de toujours avoir la version du code la plus à jour. Voici le lien de notre projet public sur GitHub :

https://github.com/Flo3171/IFB_projet_Belote

(Flo3171 est le pseudo de Florian CLOAREC et Fituning35 celui de Carlo AZANCOTH)

Solutions techniques et fonctionnement du programme

Nous allons maintenant vous détailler les solutions techniques que nous avons mises en place et la façon dont nous avons réalisé les différents points du cahier des charges dans l'ordre chronologique ou cela a été implémenter dans le projet.

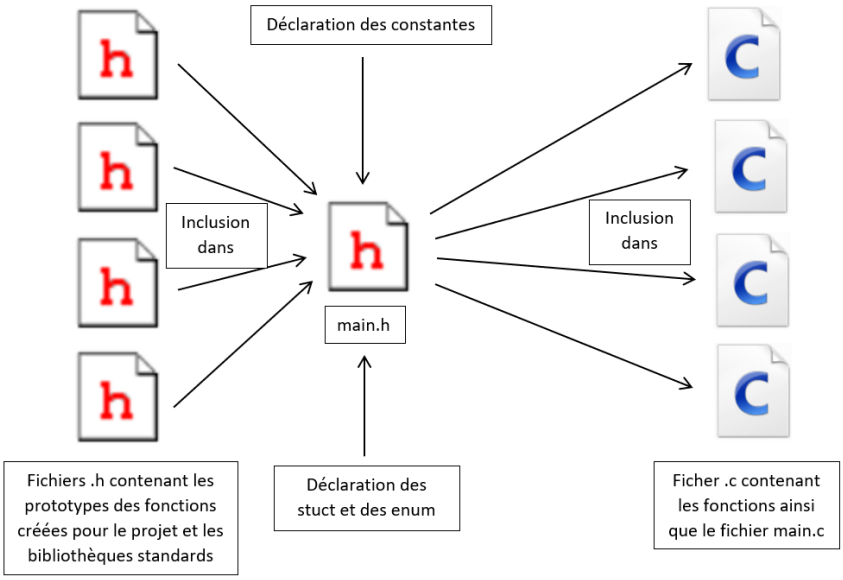
Modularité

Dès le début du projet nous avions conscience que le projet allait être composé d'un grand nombre de fonctions et de lignes de code, c'est pourquoi afin d'avoir un projet clair et ordonné, nous avons décidé de séparer les différentes fonctions dans des fichiers séparés ; à l'intérieur de ces fichiers les fonctions sont regroupées selon un thème commun (affichage, formatage, gestion des cartes, gestion des fichier…). Afin d'inclure les prototypes des fonctions à tous les endroits où cela est nécessaire, nous avons créé un fichier main.h ; ce fichier regroupe toutes les constantes, les énumérations est les structures utilisées dans ce projet, mais c'est surtout dans ce

fichier que sont inclus tous les fichiers .h associés à chaque fichier .c contenant nos fonctions.

Ce fichier main.h est alors inclus au début de chaque fichier contenant les fonctions du projet. Ainsi cela permet de s'assurer que toutes les fonctions créées spécifiquement pour ce projet ainsi que tous les autres objets susceptibles d'être manipulés par les fonctions soient utilisables, qu'importe le fichier dans lequel elles se trouvent. C'est aussi dans ce fichier main.h que nous avons inclus les bibliothèques standards que nous utilisons dans le programme. Nous avons bien conscience que cette solution n'est pas la plus optimale. En effet dans certains fichiers, des portions de code sont incuses alors qu'elles ne seront jamais utilisées, ce qui augmente de façon non négligeable la taille de l'exécutable du programme. Mais nous avons choisi cette solution car elle permet une grande liberté lors du développement du projet. En effet avec autant de fichiers, on est amené à passer très souvent d'un fichier à l'autre. Grâce à cette solution, lorsque l'on veut rajouter une fonction ou simplement appeler une autre fonction dans une fonction, il n'est pas nécessaire de s'assurer que le prototype de la fonction est bien inclus dans le fichier dans lequel on travaille. Cela permet de se concentrer sur ce que l'on fait, de gagner du temps et surtout d'éviter de nombreuses erreurs de compilations. Néanmoins cette manière de faire entraîne un autre problème : avec autant de fichiers inclus les uns dans les autres, il est alors possible que par inadvertance on crée une boucle d'inclusion infinie, ce qui empêcherait la compilation de se dérouler normalement. C'est pourquoi chacun des fichiers .h est entouré d'un code préprocesseur qui permet de ne compiler le contenu du fichier seulement si c'est la première fois qu'il est lu par le préprocesseur.

```
/*Code utilisé pour protéger le programme des boucles d'inclusions infinie*/
#ifndef _NOM_FICHIER_H_
#define _NOM_FICHIER_H_
/*contenu du fichier*/
#endif /* _NOM_FICHIER_H_ */
```



Les types de variable personnalisée

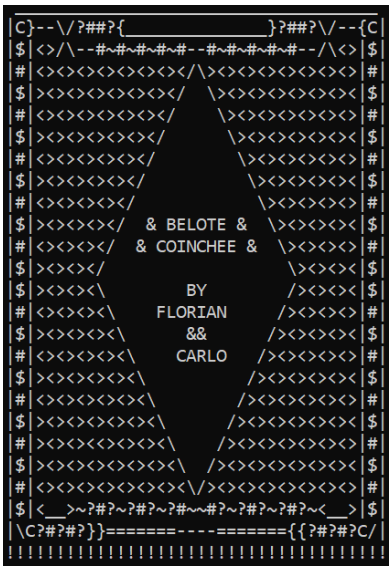
Dans ce projet un grand nombre de données à stoker en mémoire n'était pas sous la forme d'un nombre afin d'éviter de devoir stocker ces informations sous forme de chaîne de caractères, ce qui aurait été très lourd à manipuler et aurait pris beaucoup de place en mémoire. Nous avons décidé pour éviter ce problème de créer des types de variable. Nous avons donc créé une enum nommée Couleur, une autre nommée Valeur puis une stcuct nommée Carte qui est composée d'une sous-variable de type Couleur et une sous-variable de type Valeur. Ainsi nous pouvions manipuler les variables de type Carte comme une unique variable et il devient alors très facile de manipuler des tableaux de variable de type Carte. Nous avons aussi créé des types de variable personnalisée pour les Contrat

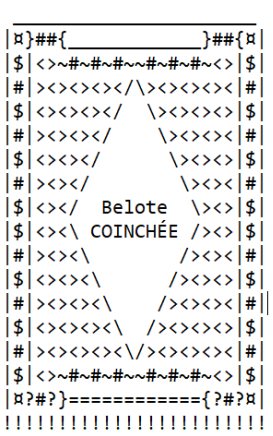
Nous avons réalisé 2 structures pour simplifier notre code et le rendre plus compréhensible, la première structure est le Type Carte qui est composée de d'une sous-variable de type Valeur (AS, ROI, DAME, …) et de type Couleur (CŒUR, CARREAU, PIQUE, TRÈFLE). La deuxième structure est de type Contrat qui prend pour composante une sous-variable de type Joueur (le joueur qui prend le contrat), une de type NbPoint (valeur du contrat), une de type Couleur (couleur de l'atout) et une de type Coinche (si le contrat à été coinché ou surcoinché)

```
typedef struct Carte
{
    Couleur couleur;/*! type Couleur, donne la famille à laquelle la carte
                        apprartient */
    Valeur valeur; /*! type Valeur, donne la valeur de la carte */
}Carte;

typedef struct Contrat
{
    Joueur preneur; /*! type Joueur, joueur qui a pris le contrat */
    NbPoint nbPoint;/*! type NbPoint, nombre de point du contrat */
    Couleur atout;/*! type Couleur, couleur de l'atout qui à été prise pour
                       le contrat*/
    Coinche coinche;/*! type Coinche, détermine si le contrat été coinché
                        ou surcoiché*/

}Contrat;
```

Les Menus

Afin que l'utilisateur puisse utiliser toutes les fonctionnalités que nous avons développées, nous avons ajouté la fonction menuPrincipal qui est l'unique fonction appelée dans le main.c. Premièrement la fonction affiche le logo (qui représente une carte), que l'on a conçu avec les 256 caractère du code ASCII étendu.





Nous avons donc réalisé 2 versions du logo, car nous nous sommes rendu compte que le premier logo était trop petit pour afficher à l'intérieur de la carte les différents sous-menus (comme sur l'image ci-dessus). Ensuite la fonction se charge d'acquérir le pseudo du joueur qui est directement enregistré dans les fichiers de sauvegarde gestion_scores_joueurs.csv. Nous arrivons ensuite au menu principal qui est composé de 7 sous-menus. Tout le code est placé dans une boucle while pour répéter indéfiniment cette partie du code qui appelle le cœur du projet, le seul moyen de sortir de la boucle est le sous-menu 7 qui ferme les fichiers précédemment ouverts puis met fin à la fonction.

Nous avons ensuite les sous-menus qui lancent une nouvelle partie qui sera jouée par l'utilisateur ou le sous-menu pour jouer 1000 parties entre les IA, cela nous a permis de récupérer des statistiques afin de pouvoir tester la capacité de nos IA à remplir des contrats et à gagner des parties. Ensuite on a les paramètres qui nous permettent de modifier les pseudos des joueurs comme on le souhaite dans la limite des caractères autorisés et le changement d'utilisateur qui permet l'acquisition d'un nouveau pseudo et d'accéder à une autre sauvegarde. Nous avons également les statistiques, qui permettent de voir les statistiques de l'utilisateur sélectionné, comme le nombre de victoires, le score maximal et le nombre de manches minimum pour gagner une partie. Pour finir le leaderboard qui nous permet de voir les meilleurs scores enregistrés.

Les pseudos

Pour les pseudos, nous avons décidé de créer un tableau de char à 2 dimensions, il est de dimension 4x21, 4 pour les 4 joueurs et 21 qui correspond à la taille maximum des pseudos, plus ''\0''. En effet en utilisant cette méthode il est plus facile d'enregistrer le nom des joueurs et de les envoyer dans les fonctions. On peut donc utiliser une variable personnalisée de type Joueur pour récupérer le pointeur vers le pseudo du joueur NORD, SUD, EST, OUEST en indiquant directement l'emplacement dans le tableau, par exemple pseudo[SUD-1] correspond au pseudo du joueur SUD.

```
/** On crée un tableau de char de dimension 4x21 **/
char pseudo[4][TAILLE_MAXI_PESEUDO+1];

/** On defini les pseudos d'IA pas défaut **/
strcpy(pseudo[NORD-1],"A_Philipe");
strcpy(pseudo[EST-1],"Gilou");
strcpy(pseudo[OUEST-1],"Tutu");
```

Utiliser cette méthode a été utile pour l'acquisition des pseudos avec la fonction acquisitionPsseudoAvecMessage et surtout la fonction paramètre qui l'appelle car on peut lui envoyer seulement un pointeur pour tous les pseudos. La fonction d'acquisition du pseudo elle demande à l'utilisateur de saisir un nouveau pseudo pour le joueur sélectionné précédemment, ensuite elle vérifie la validité de du pseudo (longueur maximale, caractères interdits, pseudo vide) et redemande le pseudo tant que la saisie est incorrecte.

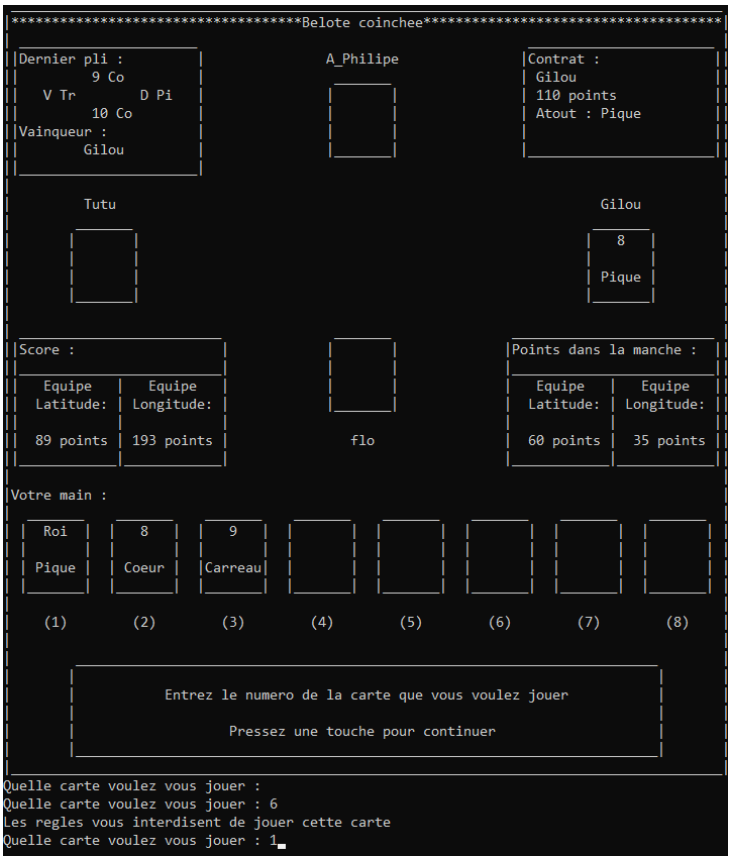
```
switch (controle)
        {
            case 1 : afficheSousMenus("Votre pseudo possede des carracteres
interdis. Choisisez votre pseudo:","pseudo");
                break;
            case 2 : afficheSousMenus("Votre pseudo n'as pas le nombre de
carracteres recquis. Choisisez votre pseudo:","pseudo");
                break;
            case 3 : afficheSousMenus("Votre pseudo ne respecte pas les
criteres. Choisisez votre pseudo:","pseudo");
        }
```

L'affichage

Nous avons décidé de développer les fonctions d'affichage dès le début du projet afin de pouvoir voir ce qui se passe lors des tests et du debug des autres fonctions. Etant donné que nous utilisons des types de variables personnalisées pour les cartes et non des chaînes de caractères, nous ne pouvions pas afficher directement le contenu de la variable tel quel. C'est pourquoi nous avons mis en place les fonctions de formatages, qui permettent de faire la convention entre une variable de type Carte (ou Contrat) et une ou plusieurs chaînes de caractères. Puis ces chaînes de caractères une fois formatées sont affichées dans la console à l'aide de la fonction printf par fonctions chargées de l'affichage.

Sachant que nous devions réaliser ce projet et qu'il n'était pas envisageable de mettre en place une interface graphique, nous avons tenté de rendre l'interface avec l'utilisateur la plus belle et la plus ergonomique. Pour ce faire nous avons créé des fonctions de formatage qui permettent de centrer les chaînes de caractères pour que le pseudo des joueurs quel qu'il soit s'affiche toujours au centre de l'espace où il est sensé s'afficher.

Nous avons de plus prévu le cas où la chaîne de caractères à afficher dépasserait la taille disponible dans le cadre où elle est affichée ; dans ce cas, la chaîne est coupée afin de ne pas décaler les autres affichages qui sont parfois sur la même ligne. Afin de formater les messages de tailles plus longues, une fonction (decoupeChaine) se charge de couper la chaîne au niveau d'un espace et d'afficher le reste de la chaîne sur la ligne suivante.



Pour afficher les Menus nous avions premièrement fait un affichage uniquement avec un printf pour le menu principal. Ensuite nous avons essayé de créer une fonction un peu plus générale afficheSousMenu pour afficher des messages dans la console, comme par exemple les instructions de jeu. Mais cette fonction était très vite obsolète lorsque nous avons réalisé les sous-Menus, en effet elle ne nous permettait pas de gérer l'espacement ou des éventuels retours. Nous avons donc fait la fonction afficheMenuSelection, celle-ci nous permettait de modifier le titre du sous-menu et de sauter des lignes en ajoutant ''\;'' à la fin de la phrase, ce qui nous a été très utile pour tous les autres affichages dans la console pour écrire un texte plus aéré et lisible ainsi qu'afficher les listes d'options dans les sous-menus.

```
printf("\t _____________________ \n");
printf("\t|¤}##{_____________}##{¤|\n");
printf("\t|$|<>Menu Principal<>|$|\n");
printf("\t|#|                   |#|\n");
printf("\t|$| 1-nouvelle partie|$|\n");
printf("\t|#|                   |#|\n");
printf("\t|$| 2-leaderboard     |$|\n");
printf("\t|#|                   |#|\n");
printf("\t|$| 3-statistiques    |$|\n");
printf("\t|$|                   |$|\n");
printf("\t|#| 4-changement       |#|\n");
printf("\t|$|   d'utilisateur   |$|\n");
printf("\t|#| 5-paramètres      |#|\n");
printf("\t|$|                   |$|\n");
printf("\t|#| 6-quitter         |#|\n");
printf("\t|$|<>~#~#~#~~#~#~#~<>|$|\n");
printf("\t|¤?#?}============{?#?¤|\n");
printf("\t!!!!!!!!!!!!!!!!!!!!!!!\n");
```

La fonction afficheInterfacePli est la fonction qui se charge d'afficher le plateau de jeu durant un pli. Elle est appelée un grand nombre de fois et c'est le résultat de cette fonction que l'utilisateur voit durant la majorité du temps d'exécution du programme. Afin de déterminer la forme et l'emplacement des différentes informations qui sont affichées à l'écran, nous avons commencé à travailler sur un fichier texte ouvert dans le bloc note ceci nous a permis de voir directement le résultat. La fonction prend comme paramètre toutes les informations qui vont devoir être affichées comme la main du joueur, les scores ou bien les cartes posées sur la table… La fonction envoie toutes ces informations aux fonctions de formatage associées puis les chaines de caractères ainsi formatées sont affichées à l'endroit voulu sur l'écran grâce à plusieurs printf(). Nous avons choisi d'utiliser un printf par ligne à afficher pour des raisons de lisibilité du code, ainsi en lisant le code il est possible de distinguer la forme de l'interface ce qui permet de faire des modifications beaucoup plus vite.

Distribution des cartes et phase d'annonces des contrats

Afin de distribuer les cartes aux différents joueurs de manière rapide et simple nous avons décidé de ne pas utiliser une méthode proche de ce qui se fait lors d'une partie de cartes réelle où on distribue une carte par joueur en faisant des tours. Nous avons préféré le faire de manière aléatoire. Pour ce faire on déclare un tableau de 32 variables de type Carte et on le remplit avec toutes les cartes d'un jeu de 32 cartes. Puis on déclare un second tableau de Carte à deux dimensions (`Carte tableauCarte[4][8];`), la première dimension vaut 4 et correspond au nombre de joueurs et la seconde vaut 8 et correspond au nombre de cartes que chaque joueur à en main. C'est ce tableau à deux dimensions qui va contenir toutes les cartes que tous les joueurs ont en main durant la partie.

Afin de s'assurer que le mélange des cartes sera bien aléatoire on se sert de le fonction rand() on obtient alors un nombre aléatoire qui sera l'indice de la carte que l'on va distribuer en premier, cette carte est alors ajoutée au premier emplacement du tableau contenant les cartes de tous les joueurs et elle est supprimée du tableau qui contient les cartes à distribuer afin qu'aucune carte ne puisse être distribuée 2 fois. On continue cette opération en incrémentant l'indice ou la carte va être distribuée jusqu'à ce que le tableau contenant les cartes des joueurs soit rempli de toutes les cartes du jeu de manière unique. Ainsi pour accéder à la carte n du joueur SUD il faudra entrer : `tableauCarte[SUD-1][n]` ce qui en termes de pointeur est équivalent à `*(pTableauCarte + (SUD-1)*8 + n).`

Une fois les cartes distribuées, on entre la phase d'annonce des contrats. La fonction qui se charge de cela appelle selon si le joueur qui doit proposer un contrat est un utilisateur ou un ordinateur la fonction associé respectivement proposeContratUtilisateur et choixCarteIA. Ces fonctions renvoient chacune une variable de type Contrat qui représente le contrat que le joueur a choisi de prendre. Si ce contrat à un nombre de points supérieur à 0, ce qui signifie que le joueur n'a pas passé alors ce nouveau contrat vient remplacer celui proposé par le dernier joueur à avoir proposé un contrat.

C'est lors de l'acquisition que l'on se charge de vérifier que le nouveau contrat proposé a bien un nombre de points supérieur au dernier contrat. Les joueurs continuent de proposer des contrats chacun leur tour jusqu'à qu'après le premier tour, 3 joueurs d'affilé passent. Une fois que plus personne ne souhaite surenchérir, la fonction qui gère les contrats se termine et revoit le contrat définitif qui sera appliqué pour le reste de la manche.

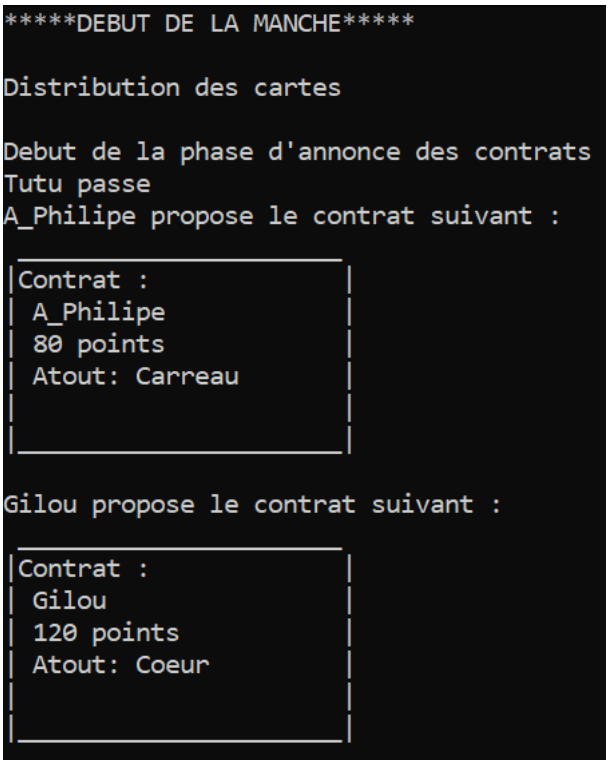
La possibilité de coincher et de sur coincher a aussi été mise en place, ainsi un joueur a la possibilité de coincher si un joueur de l'équipe adverse a proposé un contrat. En coinchant, le joueur propose un nouveau contrat identique au précèdent à la différence que la sous variable .coinche ne vaut plus NORMAL, mais vaut alors COINCHE. Un joueur a la possibilité de sur coincher si un joueur de l'équiper adverse a coinché un contrat proposé par un membre de son équipe. La sur coinche se passe comme pour la coinche à la différence que la sous variable .coinche vaut alors SURCOICHE.

Calcul de la force d'une carte

Afin de déterminer quelle carte est la plus forte et va remporter le pli, la fonction forceCarte permet d'associer à chaque carte une valeur numérique comprise entre 0 et 1 qui reflète ca force, plus ce nombre sera grand plus la carte sera forte. Après avoir cherché qu'elle serait la méthode la plus efficace et la plus pratique afin de donner une force à chaque carte, nous avons décidé de passer par une approche probabiliste.

Ainsi la couleur de l'atout (qui peut être sans-atout et tout-atout) est et la couleur de l'entame (la première carte jouée dans le pli) sont des paramètres de la fonction forceCarte. La force d'une carte se calcule avec la formule suivante : $\frac{nombre\ de\ cartes\ battues}{nombre\ de\ cartes\ total-1}$. Ce qui signifie par exemple que le valet d'atout va battre les 31 autres cartes du paquet et ainsi il aura une force de 1, à l'inverse le 7 de non-atout s'il n'est pas dans la couleur de l'entame va battre aucune carte et aura donc une force de 0. Avoir fait des enum et des struct pour définir la variable personnalisée Carte nous permet de manipuler les sous variables .couleur et .valeur comme des entiers et ainsi nous pouvons utiliser des switch qui dans le cas de cette fonction améliorent grandement la lisibilité du code. Grâce à cette fonction, il devient très facile de trouver le vainqueur d'un pli, il faut alors chercher quelle carte a la force maximum parmi les cartes du pli. Mais cette fonction a aussi d'autres usages, elle est utilisée dans les fonctions ia, afin d'évaluer et de pouvoir faire des calculs dessus afin de prendre une décision.

Intelligence Artificielle (ia)



Afin de réaliser ce programme, nous avons dû créer deux ia. La première permet de déterminer quelle sera le contrat que l'ordinateur va prendre ou s'il va passer et la seconde permet de choisir quelle carte l'ordinateur va jouer lors de chaque pli. Ces deux ia prennent la forme de deux fonctions (proposeContratIA et choixCarteIA) qui prennent comme paramètre de nombreuses informations sur la partie comme les cartes que le joueur à en main et retournent respectivement une variable de type contrat et un entier qui correspond à la position de la carte à jouer dans la main du joueur.

Afin de déterminer quel contrat l'ordinateur va prendre, le programme fait la somme de la force de chacune des cartes dans toutes les couleurs possibles (y compris tout-atout et sans-atout) et regarde dans quelle couleur la valeur est la plus grande, une grande valeur correspond à une main forte dans la couleur. En faisant des tests sur cette méthode, nous avons découvert que la somme des forces des cartes de la main était plus grande en tout atout que dans les autres couleurs. Afin de régler ce problème, nous avons remis cette valeur à l'échelle, ainsi la plus forte main en tout-atout vaut la même valeur que la plus forte main dans une autre couleur et le nombre que l'on obtient est compris entre 0 et 1. Puis la fonction regarde si ce nombre dans la meilleure couleur est supérieur au seuil minimum de prise de contrat. Si tel est le cas, l'ordinateur va chercher à prendre un contrat dans cette couleur.

La valeur du contrat que l'ordinateur va prendre est proportionnelle au nombre trouvé précédemment plus ce nombre sera supérieur à la valeur du seuil minimum de prise de contrat, plus l'ordinateur prendra un contrat avec un nombre important de points. Afin de respecter les règles du jeu, la fonction va alors tester si le contrat quelle veut proposer est valide et pour se faire on teste, si le nombre de points proposés par ce nouveau contrat est supérieur au dernier contrat proposé (qui est celui avec le plus grand nombre de point sachant que les autres joueurs doivent aussi surenchérir), alors la fonction renvoie une variable de type contrat avec le nombre de points et la couleur qu'elle a déterminée. Sinon si sa main ne lui permet pas de dépasser le seuil minimal de prises alors elle passe, ce qui se matérialise par le renvoi par la fonction d'une variable de type Contrat dont le nombre de points vaut 0.

Si le joueur ne peut pas surenchérir car un membre de l'équipe adverse a proposé un contrat supérieur au maximum de ce qui le joueur peut enchérir, alors il va regarder s'il peut coincher. Nous avons déterminé par des tests sur un grand nombre de parties deux paramètres seuilMiniCoinche et deltaCoinche. Alors le joueur va coincher si la valeur du contrat proposé par le dernier joueur est supérieure à seuilMiniCoinche et que la différence entre la valeur du dernier contrat et la valeur du contrat qu'il aurait proposé est inférieure à deltaCoinche.

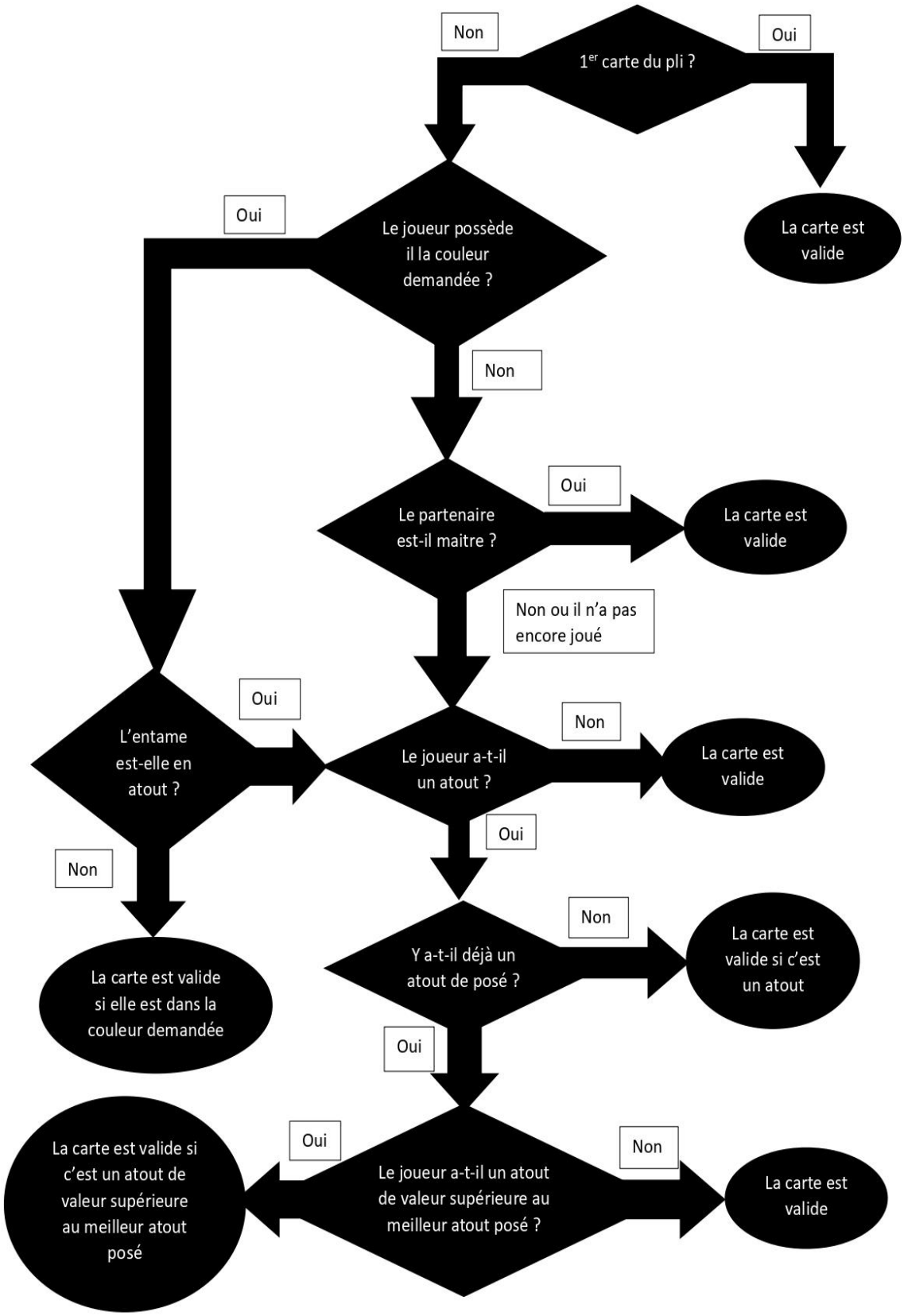
Pour déterminer quelle carte l'IA peut jouer nous avons tout premièrement fait une version pour le débug, c'est-à-dire que pour déterminer quelle carte l'IA va jouer nous parcourons toute sa main et dès que la carte est jouable la fonction renvoie le numéro de la carte à jouer à la fonction poseCarte qui s'occupera de jouer la carte. Ensuite nous avons rajouter un niveau 2 pour l'IA pour respecter le cahier des charges. Ce niveau d'IA (niveau 2), va donc jouer un peu plus stratégiquement, le code de cette partie est divisé en 3 étapes. La première étape était de vérifier quelles cartes étaient jouables et de les enregistrer dans un nouveau tableau, le but étant de pouvoir traiter ce tableau par la suite.

La deuxième étape était d'analyser les cartes posées dans le pli et de voir si dans les cartes qui étaient jouables une des cartes permette de gagner le pli et de la même manière qu'avant d'enregistrer les cartes qui permettraient de gagner. Dans la troisième partie on vérifie si l'on peut gagner le pli, si on ne peut, pas ou si une seule carte est possible à jouer, le but est par exemple si on ne peut pas gagner le pli que la carte la plus faible soit jouée ou dans le cas contraire on joue la meilleure carte possible. Finalement la fonction renvoie uniquement le numéro de la carte à jouer. De plus nous avons ajouté une sécurité au cas où le numéro de carte n'est pas valide, nous testons donc si le numéro de carte est valide et si il ne l'est pas nous passons au niveau 1 d'IA pour que la fonction renvoie toujours un numéro jouable pour que la fonction poseCarte marche.

Déterminer si on a le droit de jouer une carte

A de nombreuses reprises lors de l'exécution du programme il est nécessaire de déterminer si la carte d'un joueur ou un ordinateur est valide ou non. Pour ce faire nous avons créé la fonction carteValide qui renvoie 1 si la carte que l'on veut jouer et valide en fonction des cartes déjà présente sur la table, des cartes dans la main du joueur et de la couleur de l'atout. Afin de transposer les règles du jeu qui ne sont pas évidentes à traduire en langage algorithmique, nous avons créé l'organigramme si dessous afin de déterminer si une carte peut être jouée, ainsi une fois ce travail préliminaire effectué, il a été beaucoup plus facile de rédiger la fonction carteValide en faisant une succession de if et de else if imbriqué les uns dans les autres.

Organigramme décisionnel de la fonction qui détermine si une carte peut être posée dans un pli



Gestion de annonces

Dans les derniers jours du projet lorsque nous avions remplit toutes les fonctionnalités demandées par le cahier des charges il nous a semblé bon de chercher à améliorer encore un petit peu notre projet en ajoutant les annonces. En effet celle si sont importante dans le jeu de la Belote coinchée et ont une incidence non négligeable sur le déroulement de la partie. Afin de détecter lorsqu'un joueur annonce une belote rebelote, on teste dès qu'une carte est posée s'il s'agit d'une Dame ou d'un Roi d'atout. Si tel est le cas, on affiche un écran un message indiquant à l'utilisateur que le joueur en question annonce une belote. Lorsque le joueur pose la seconde carte, on affiche également un message indiquant qu'il annonce une rebelote. A la fin de la manche, lorsque tous les plis sont terminés, on donne 20 points au joueur qui ont annoncé une belote/rebelote durant la manche.

Afin de déterminer si certain des joueurs ont d'autres annonces à faire, on appelle la fonction rechercheAnonce au début de la manche. Cette fonction recherche dans la main de chaque joueur s'il y a une des annonces décrites dans l'image si contre. Si le joueur possède une de ses annonces, alors le programme affiche un message pour indiquer que le joueur fait une annonce et ajoute le nombre de point associé au point d'annonce du joueur.

Gestion des scores

Pour cette partie du projet nous avons fait une fonction pointPli qui compte les points à chaque pli. Cette fonction prend en paramètre le tableau contenant les cartes jouées pendant le pli et utilise des switch pour trouver la bonne valeur de la carte puis l'additionne au total. Tout d'abord nous vérifions quel atout est joué, SANS_ATOUT, TOUT-ATOUT ou ATOUT (cœur, carreau, pique, trèfle), dans les 2 premiers cas les cartes valent toutes la même valeur, en revanche dans le dernier cas il fallait prendre en compte si la carte était de la couleur de l'atout pour lui attribuer une valeur.

```
switch(atout){
            case TOUT_ATOUT:
                    {
                    switch (pli[i].valeur){
                            ...
                        }
                    }
                    break;
            case SANS_ATOUT:
                    {
                     switch (pli[i].valeur){
                            ...
                        }
                    }
                    break;
            default:
                    {
                    if (pli[i].couleur == atout){
                        switch (pli[i].valeur){
                            ...
                        }
                    }else{
                        switch (pli[i].valeur){
                            ...
                        }
                    }
                }
                break;
        }
```

Poser la carte

La pose de carte s'effectue par la fonction poseCarte, pour cela on lui envoie le pointeur vers la main du joueur, le pointeur vers les cartes du pli et le numéro de la carte à jouer. Premièrement la carte à jouer est récupérée de la main du joueur, et enregistrée dans une variable de type Carte. Dans un second temps la carte est supprimée de la main du joueur et les autres cartes sont repositionnées par la fonction supprime carte. Finalement la carte est ajoutée dans le tableau des cartes du pli à la position du joueur.

```
int poseCarte (Joueur joueur,int numCarte, Carte *pMainJoueurs, Carte
pli[],int carteRestante)
{
    int retour=NULL;
    Carte carteAJouer;
    carteAJouer =  *((numCarte-1)+ pMainJoueurs);

    supprimeCarte(pMainJoueurs,carteRestante,numCarte-1);
    setCarte(pli+joueur-1,carteAJouer.valeur,carteAJouer.couleur);
    retour=1;
    return retour;
}
```

Sachant que nous utilisons un type de variable personnalisée pour les cartes, si l'on veut attribuer une valeur et une couleur à une carte, cela doit se faire en deux étapes. Afin de simplifier cela nous avons créé la fonction setCarte qui affecte en une fonction la valeur et la couleur souhaitée à une carte.

Pour pouvoir manipuler les tableaux de carte qui sont utilisé de nombreuses fois dans le programme nous avons créé la fonction supprimeCarte qui qui permet de supprimer de manière intelligente une carte dans un tableau. Pour ce faire on passe en paramètre l'indice dans le tableau de la carte de la carte à supprimer, la fonction va alors copier (grâce à la fonction set carte) un indice plus haut toutes les cartes dont l'indice initiale est supérieur à celui de la carte à supprimer, la dernière carte du tableau est alors remplacée par une carte vide pour éviter les doublons. Ainsi la carte à supprimer sera écrasée et il n'y aura pas de trou à l'intérieur du tableau, on rajoute seulement une carte vide à la fin.

Gestion des fichiers

Nous avons décidé pour la sauvegarde de tous les scores et des meilleurs scores de diviser le travail en faisant 2 fichiers séparés. Nous avons donc fait un fichier pour les statistiques des utilisateurs ou l'on sauvegarde le nombre de victoires, le maximum de points et le nombre de manches pour gagner une partie, et un fichier pour enregistrer les 10 meilleurs scores. Nous avons choisi de définir des constantes pour la position de chaque élément, ainsi pour se déplacer dans le fichier on utilise dans la fonction fseek() NB_CARRACTERE_SCORE pour se déplacer dans le fichier en le multipliant par le numéro de ligne, on a donc dû fixer le nombre de caractères par ligne pour pouvoir se déplacer de cette manière

```
#define NB_CARRACTERE_SCORE 33
#define POSITION_NB_VICTOIRE 21
#define POSITION_SCORE_MAX 25
#define POSITION_NB_MANCHES_POUR_GAGNER 30
#define NB_CARRACTERE_LEADERBOARD 26
#define POSITION_RECORD_VICTOIRE 21
```

La structure des fichiers est de telle sorte à ce que tous les éléments enregistrés soient complets tout en fixant le nombre de caractères par ligne, ainsi on contrôle à l'aide des ''%20s'' et ''%4d'' la taille des information écrites dans le fichier.

Structure du fichier :

- Statistiques personnelles : <pseudo> (%20s) ; <nombre de victoires> (%3d) ; <score maximal> (%4d) ;<nombre de manche pour gagner> (%1d)
- Leaderboard : <pseudo> (%20s) ; <nombre de victoires> (%3d)

```
pseudos;meilleurs scores
flo                 ;  0
Carlo               ;  0
Typhie              ;  0
Sroyce              ;  0
Thorgaran           ;  0
Hide                ;  0
JEEEEEEEEEEEEEEEEEJ;  0
Geko68              ;  0
Glaeith             ;  0
supercelianh        ;  0
```

	A	B
1	pseudos	meilleurs scores
2	flo	0
3	Carlo	0
4	Typhie	0
5	Sroyce	0
6	Thorgaran	0
7	Hide	0
8	JEEEEEEEEEE	0
9	Geko68	0
10	Glaeith	0
11	supercelianh	0

Nous avons utilisé des fichiers .csv car ils nous permettent d'enregistrer corectement les informations à stocker et de pouvoir lire ces informations avec un tableur (excel).

La première fonction s'occupe donc d'enregistrer les utilisateurs, pour cela elle recherche dans le fichier le pseudo du joueur, 2 cas sont donc possibles soit le joueur n'existe pas il est donc rajouté à la fin du fichier, soit le joueur existe dans ce cas la lecture s'achève. Dans tous les cas cette fonction renvoie le numéro de ligne à laquelle sont notées les statistiques de l'utilisateur comme ça on n'est pas obligé de relire tout le fichier pour retrouver le pseudo et enregistrer les valeurs.

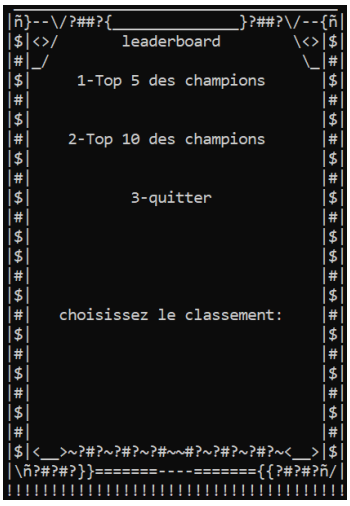
La deuxième fonction permet à partir la ligne reçue par la fonction précédente de modifier les différentes statistiques. La fonction a donc été codée de façon à pouvoir modifier une des statistiques À la fois pour éviter de lui envoyer 3 paramètres de statistiques à la fois, ainsi elle prend un paramètre type qui nous permet donc de sélectionner si l'on souhaite changer le nombre de victoire, le score, ou le nombre de manches. De plus cette fonction renvoie le nouveau score qui vient d'être enregistré, par exemple si on envoie 1 à la fonction dans le mode d'enregistrement de victoire, elle va ajouter à l'ancien score la nouvelle victoire et renvoyer le nombre de victoires total de l'utilisateur, cela est utile pour la fonction suivante.

La dernière fonction essaye d'inscrire le nouveau score du joueur dans la liste des meilleurs scores qui se trouvent dans le second fichier. Pour cela la fonction commence par chercher si le pseudo existe déjà dans le fichier, puis il sauvegarde la ligne et commence à faire un tri à bulle à partir de cette position pour essayer de remonter le joueur, si le pseudo n'est pas trouvé par défaut le tri À bulle commence à partir de la 10ème ligne.

Affichage des statistiques

Pour l'affichage des statistiques Nous avons réalisé 2 fonctions, une pour les statistiques personnelles et une pour le leaderboard. Le principe des 2 fonctions est semblable, dans un premier temps elles vont extraire les informations du fichier et les concaténer dans une chaine avec une mise en forme, on rajoute donc un peu de texte pour agrémenter et des '' ;'' pour sauter des lignes. Le but de cette opération est de créer un seul tableau de char à envoyer à la fonction afficheMenuSelection.

```
for(int ligne=0 ; ligne<10 ; ligne++){
/** Ajout de la position du joueur **/
    strcat(listeDesScores,"#");
    itoa(ligne+1,score,10);
    strcat(listeDesScores,score);
    strcat(listeDesScores," ");

/** Récupération puis ajout du pseudo du joueur à la suite **/
    fseek(fichier,ligne*NB_CARRACTERE_LEADERBOARD,SEEK_SET);
    fscanf(fichier,"%20s",pseudo);

/** rajoute "..." à la fin du pseudo pour les pseudos les plus longs **/
    if(strlen(pseudo)>14){
        fseek(fichier,ligne*NB_CARRACTERE_LEADERBOARD,SEEK_SET);
        fscanf(fichier,"%14s",pseudo);
        strcat(listeDesScores,pseudo);
        strcat(listeDesScores,"...");
    }else{
        strcat(listeDesScores,pseudo);
    }
    strcat(listeDesScores," score:");

/** Récupération et ajout du score du joueur **/
fseek(fichier,ligne*NB_CARRACTERE_LEADERBOARD+POSITION_RECORD_VICTOIRE,SEEK
_SET);
    fscanf(fichier,"%s",score);
    strcat(listeDesScores,score);

/** Ajout du caractère de saut de ligne **/
    listeDesScores[strlen(listeDesScores)]=';';
}
```



Ensuite cela diffère entre les 2 fonctions, la fonction pour les statistiques perso elle envoie directement la chaine de caractères dans la fonction afficheMenuSelection. En revanche dans la fonction pour le leaderboard on crée un switch qui permet à l'utilisateur de choisir entre le top 5 ou top 10 des meilleurs joueurs, ce qui va changer le nombre de lignes à sauter.

Tri des cartes dans la main

Afin de rendre plus lisible par l'utilisateur les carte de ca main nous avons réalisé une fonction (trieCarte()) qui permet de trier les carte de la main du joueur. Nous nous sommes mis d'accord sur le meilleur ordre de carte à adopter et nous avons décidé de mettre en place l'ordre suivant de gauche à droite, les carte de la famille de l'atout de la plus forte à la moins forte, puis les cartes triées par famille de la plus forte à la moins forte selon l'ordre de non-atout. La couleur de l'atout au cœur de la manche est un paramètre de la fonction trieCarte, ce qui permet de trier les cartes selon le contrat prit dans la manche. Pour faire ce tri, nous avons utilisé la méthode du tri a bulle car c'est celle qui semblait la plus adapté pour trier un petit tableau. Cette fonction de trie est appelée seulement 2 fois par manche, un première fois avant la phase d'annonce des contrats elle trie les cartes de l'utilisateur en comme si le contrat pris valait tout atout, ainsi l'utilisateur peut voir directement lequel de ces cartes sont forte avec l'ordre de l'atout. Le second endroit où cette fonction est appelée, c'est après la phase d'annonce des contrat les cartes sont à nouveau triées en fonction du contrat qui vient d'être choisi. Après réflexion, nous nous sommes rendu compte qu'il n'était pas nécessaire de trier à nouveau les carte à chaque fois que l'utilisateur pose une carte car en enlevant simplement une carte de la main les cartes de la main restaient triées.

Test du programme

Acquisition sécurisée d'entiers

Lors des tests du programme nous nous sommes rendus compte que le programme plantait lors de l'acquisition d'un entier Naturel si l'utilisateur tapait des caractères autres que de simples chiffres. Nous avons donc dû penser un algorithme pour éviter ce genre de problèmes, C'est dans la fonction acquisitionEntierSecurise qui s'occupe de l'acquisition d'entier.

Cette fonction va premièrement vérifier si le premier caractère ne soit pas un 0, nous avons dû faire cette manipulation car la fonction qui transforme un char en int renvoie un zero s'il trouve un caractère autre qu'un chiffre, c'est pour cella que dans le else si la fonction renvoie un 0 on donne une valeur de -1 à NB pour qu'on puisse différencier la saisie d'un 0 et d'une lettre. La fonction renvoie donc la valeur saisie elle remplace donc le scanf() dans toutes nos fonctions d'acquisition.

```
int acquisitionEntierSecurise()
 {
     char num[50];
    int NB=0;
/** Lecture de l'entrée du joueur de type char **/
    fgets(num,50,stdin);
/** Test si le premier caractère est un 0 **/
    if(num[0]=='0'){
        NB=strtol(num,NULL,10);
    }else{
/** Conversion du char en int **/
        NB=strtol(num,NULL,10);
        if(NB==0){
            NB=-1;
        }
    }
    return NB;
}
```

Utilisation du débuggeur GDB

Dès lors que nous avons commencé à avoir des fonctions ou des enchainements de fonctions assez complexe, il est devenu plutôt difficile de trouver d'où venait les problèmes et de les corriger rapidement. Pour ce faire nous avons mis en place le débuggeur GDB afin de pouvoir chercher les erreurs et comprendre ce qui ne marchait pas. Grâce aux break points et au wacht, nous avons été capable de suivre lors de l'exécution la valeur des variables qui nous intéressait. Ainsi en exécutant pas à pas les zones du programme qui posait des problèmes et en analysant ce qui se passait en mémoire grâce au débuggeur nous avons pu sans perdre trop de temps précieux résoudre tous les problèmes liés à l'exécution du code.

Jouer des partie ente 4 ordinateurs

Afin de gagner du temps lors du débug nous avons décidé de permettre à notre programme de réaliser des parties sans aucune interaction de l'utilisateur et entre 4 ordinateurs. Ainsi si on lance la fonction nouvellePartie() avec la paramètre utilisateur qui vaut SANS_JOUEUR alors une partie ce lance sans aucun joueur. Dans ce, si la constante DEBUG_MODE vaut 1 alors chaque action de la partie sera décrite par une courte phrase dans la console et la partie se joue en quelques millième de seconde. Si la constante DEBUG_MODE vaut 0 alors la partie se déroule sans que rien ne s'affiche à l'écran mais la partie se joue quand même. Grâce à cette fonctionnalité il est alors possible de jouer 1000 parties automatiquement. Nous avons pu remarquer que l'exécution de 1000 partie sans aucun affichage prenait en moyenne 0,5 secondes, ce qui montre que notre programme est bien optimisé.

De plus en jouant ces 1000 on récupère des statistiques notamment le pourcentage de victoire de chaque joueur et le pourcentage de contrat remplis. Grâce à ces informations nous avons pu tester et améliorer nos IA afin de déterminer lesquelles étaient les meilleures. C'est avec cette méthode que nous avons pu déterminer avec précision les valeurs que devait utiliser les fonctions d'IA afin de gagner dans le plus grand nombre de cas. Lors du débug du programme nous avons utilisé le mode de jeux entre 4 IA un grand nombre de fois afin de vérifier que les parties se déroulait conformément aux règles, que les IA jouaient bien la carte la plus adaptée et que le programme ne rencontrait pas d'erreur. Ainsi en faisant un grand nombre de partie de cette manière nous avons découvert des situations ou les règles n'était pas respecter, comme par exemple le fait de devoir jouer dans la couleur de l'atout si l'entame est en atout et que l'on ne peut pas surcouper. Nous avons pu corriger ces défauts jusqu'à ce que le programme fonctionne dans tous les cas de figure (même tout atout et sans atout).

Points d'amélioration

Afin d'améliorer notre programme nous pourrions chercher à optimiser la vitesse d'exécution, dans de nombreuse fonction il est possible de d'optimiser le code et ainsi avoir un programme plus rapide. Nous pourrions aussi tenter d'améliorer les ia qui ne joue pas toujours aussi bien qu'un humain bien entrainé, pour ce faire il faudrait que l'IA stocke toutes les cartes qui ont déjà été jouée au cours de la manche et quelle décide quelle carte joueur en fonction. Nous pourrions aussi améliorer l'affichage de notre programme en utilisant une interface graphique par exemple ou en nous servant des caractère unicode pour que les bordure de nos tableaux soient plus belle. Mais par faute de temps, nous avons préféré nous concentrer sur les fonctionnalités qui nous était imposées par le cahier des charges.

Conclusion

Expérience acquise

En conclusion, nous sommes fiers du travail que l'on a fourni et nous avons beaucoup appris et progressé en réalisant ce projet. Nous avons vu de nombreux aspects de la programmation en C, mais également d'autres ressources comme Github pour pouvoir travailler à plusieurs sur le projet. Ce travail à été très enrichissant au niveau du travail de groupe, on a dû apprendre à se partager le travail et à travailler en harmonie pour éviter les éventuels problèmes de compatibilité entre nos fonctions. De plus nous avons appris à structurer un projet clair en triant dans différents fichiers nos fonction et en commentant leur fonctionnement avec des briefs ou des commentaires directement dans la fonction. Nous avons ainsi pu apprendre à réaliser une documentation propre et détaillée de notre code.

Bilan cahier de charges

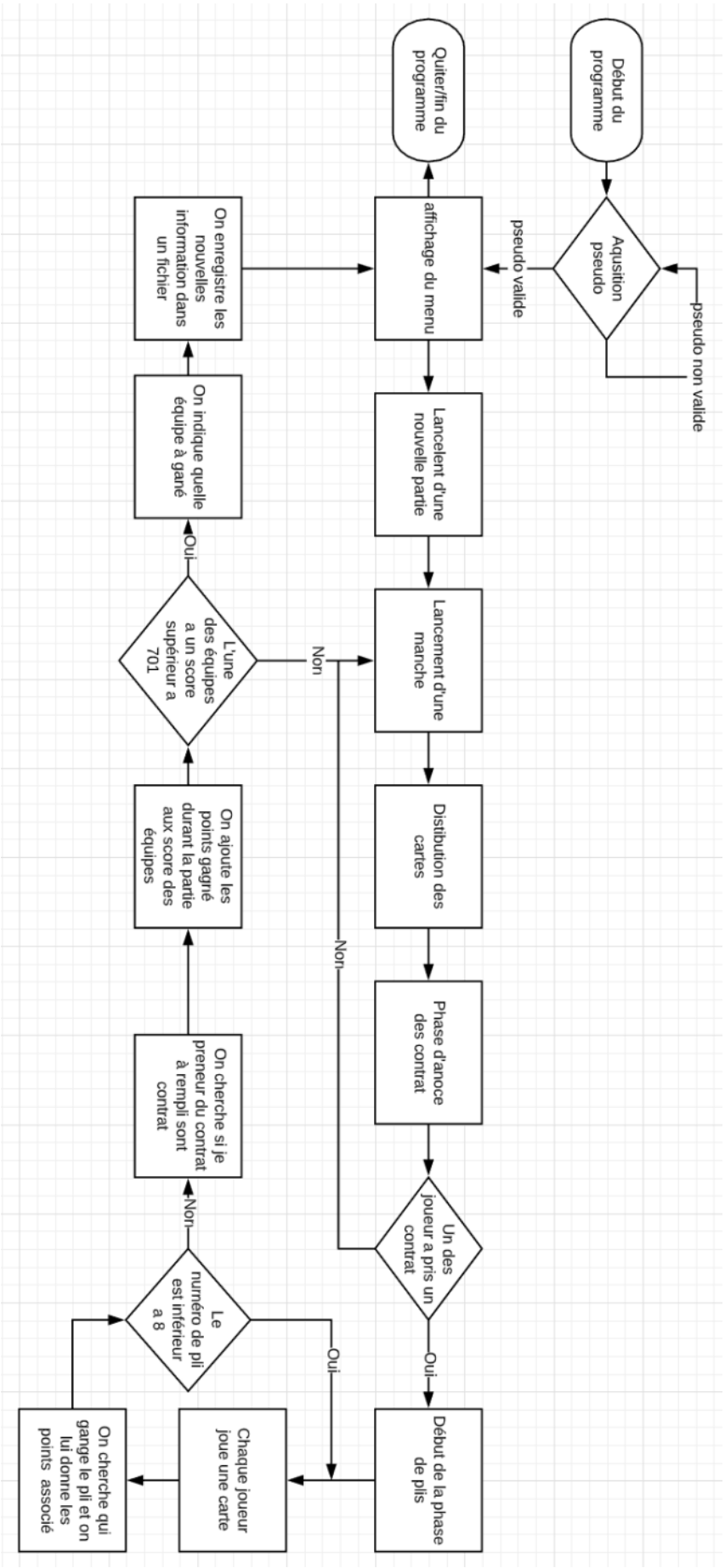
Nous avons réussi à respecter toutes les lignes du cahier des charges. Tous les affichages demandés ont été effectués, nous avons de plus ajouté un affichage en jeu du dernier pli et des points de chaque équipe. Pour le menu nous avons ajouté des fonctionnalités supplémentaires comme l'affichages des statistiques de l'utilisateur, le changement d'utilisateur, les paramètres pour changer le pseudo des IA, et la fonctionnalité pour jouer 1000 parties entre les IA.

Pour les règles de la coinche, nous avons respecté toutes les règles de La fédération française de belote, les cartes sont donc vérifiées avant d'être jouée et elles sont distribuée équitablement entre les joueurs, le calcul des points ainsi que la vérification du contrat sont faits pendant toute la partie. De plus nous avons réalisé la gestion des annonces.

Pour les IA nous avons respectée les règles de jeu indiquées dans le cahier des charges. Pour les contrats nous avons amélioré leur prise de décision par rapport à ce qui était demandé, cela à été permis par notre algorithme de calcul de puissance d'une carte. L'IA en jeu respecte les les condition indiquée, nous avons seulement ajouté 2 niveau de difficulté différents pour leur prise de décision, un niveau moyen et plus faible.

Remerciements

Nous remercions les créateurs du cours de programmation en C oppenclassroom's qui nous a grandement aides dans ce projet. Nous remercions également l'ensemble des forums qui nous ont permis d'obtenir des réponses à nos problèmes et de comprendre nos erreurs. Un grand merci aux enseignants de l'UV IFB pour leurs cours et leurs exercices formateurs. Et des remerciements spéciaux à Victor.G, Célian.H et Louis.H pour leurs conseils et les débats sans fin à propos de ce projet.


Début du programme
Aqusition pseudo
pseudo non valide
pseudo valide
Quiter/fin du programme
affichage du menu
On enregistre les nouvelles information dans un fichier
Lancelent d'une nouvelle partie
On indique quelle équipe à gané
Lancement d'une manche
Oui
L'une des équipes a un score supérieur a 701
Non
On ajoute les points gagné durant la partie aux score des équipes
Distibution des cartes
Phase d'anoce des contrat
Non
Un des joueur a pris un contrat
On cherche si je preneur du contrat à rempli sont contrat
Oui
Non
Le numéro de pli est inférieur a 8
Oui
Début de la phase de plis
Chaque joueur joue une carte
On cherche qui gange le pli et on lui donne les points associé