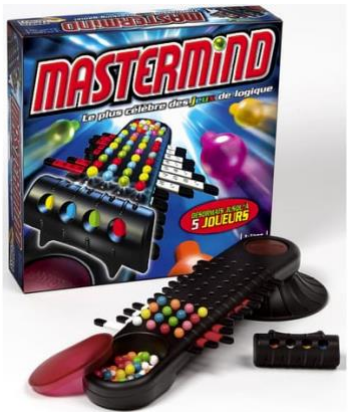
PROJET LP25

MasterMind

Theo Bedez, Antoine Planche, Paul Desandes, Thomas Duvinage

TABLE DES MATIERES

DESCRIPTION DU PROJET :



Le projet que nous avons choisi est un MasterMind. Le MasterMind est un jeu de logique, ce jeu de société est un classique et est considéré comme ludique et excellent pour développer ses capacités de logique.
Le but du jeu du Mastermind est pour l'un des joueurs d'élaborer une combinaison difficilement déchiffrable et pour son adversaire, de deviner en un minimum de coup cette combinaison.
Le MasterMind est un jeu qui se joue à 2 personnes.

LES REGLES DU JEU

Dans un premier temps, il faut choisir qui des deux personnes va élaborer la combinaison secrète. Une fois que cela est fait, le joueur désigné élabore une combinaison secrète.
Ensuite vous devez vous mettre d'accord sur le nombre de manche que vous souhaitez disputer pour vous assurer d'un vainqueur.
Une fois ces étapes réalisées le jour qui doit deviner la combinaison peut commencer à jouer.
Pour cela, il doit insérer 4 pions dans la première ligne située en bas de la grille.
Si la combinaison présentée est incorrecte :

- Le joueur qui a élaboré la combinaison secrète utilise les languettes situées sur les côtés du plateau.
  Si dans la proposition, un ou plusieurs pions de couleurs sont bien dans la combinaison mais pas à la bonne place, le joueur doit alors tirer la languette blanche selon le nombre.
  Si dans la proposition, un ou plusieurs pions de couleurs sont bien dans la combinaison et à la bonne place, le joueur doit alors tirer la languette rouge selon le nombre. Le joueur devant deviner la combinaison continue ainsi en proposant sur la seconde ligne une autre proposition, en prenant bien entendu en compte les indications des languettes rouges et blanches. Il a le droit à 12 propositions pour déchiffrer le code.

Si la combinaison est correcte :

- Le joueur qui a élaboré la combinaison secrète révèle le code et la manche est terminée. Les rôles sont alors inversés et la seconde manche peut ainsi commencer.

LE PROJET

Notre projet étant une implémentation en langage C du MasterMind, l'utilisateur aura plusieurs possibilités.

- Jouer contre un adversaire Humain
- Jouer contre l'ordinateur

JOUER CONTRE UN ADVERSAIRE HUMAIN

Dans cette configuration du jeu, l'adversaire choisit une combinaison secrète à l'abris de la personne qui doit trouver la combinaison. Dans un second temps le joueur doit retrouver la combinaison. Après chaque insertion de ligne l'adversaire indique dans le terminal si la combinaison est correcte ou non en respectant les règles mentionnées ci-dessus.

JOUER CONTRE L'ORDINATEUR

Dans cette configuration, l'ordinateur détermine aléatoirement la combinaison secrète, Puis le joueur doit retrouver cette combinaison de la même manière qu'une partie contre un Humain à la seule différence que ça sera l'ordinateur qui indiquera la validité de la combinaison donnée par le joueur.

LES OPTIONS

Durant une partie le joueur a la possibilité de :

- Enregistrer la partie en cours
- Reprendre une partie
- Supprimer une ancienne partie

REPARTITIONS DES TACHES

Pour la répartition des tâches de ce projets nous avons fait le choix de nous diviser en 2 sous-groupes. Un groupe est composé de Theo Bedez et Paul Desandes et l'autre de Antoine Planche et Thomas Duvinage.
Theo Bedez et Paul Desandes s'occuperont de l'enregistrement de partie et en cours et la reprise d'une partie et Antoine Planche et Thomas Duvinage s'occuperont des différents modes de jeu.
Vous trouverez dans la figure ci-dessous la description des tâches de chacun d'entre nous.

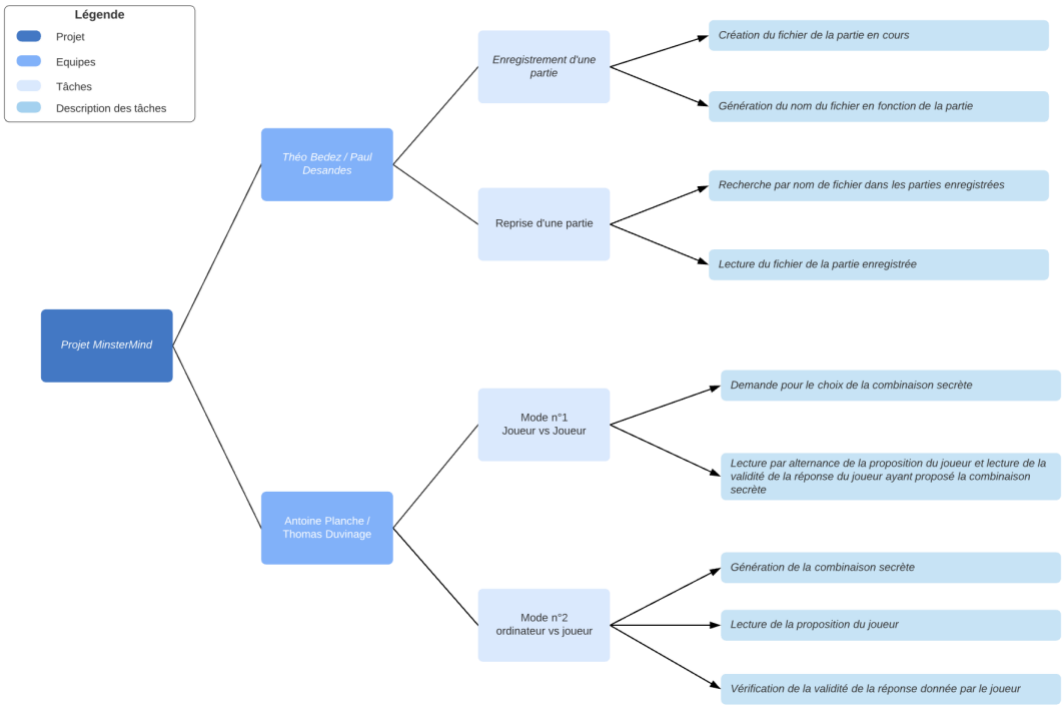


FIGURE 1 : REPARTITION DES TACHES

Afin de travailler sur le projet dans une totale collaboration et à distance, nous avons mis en place un GitHub qui nous permet de suivre l'avancée du projet et de travailler ensemble sans avoir à s'envoyer par mail ou autres moyens d'échanges les codes.

PROGRAMMATION

LES STRUCTURES

COULEUR

Pour réaliser ce projet nous avons opté pour la structure suivante :

```
typedef enum couleur {
ROUGE,
JAUNE,
BLEU,
ORANGE,
VERT,
```

```
BLANC,
VIOLET,
ROSE,
NONE
} Couleur;
```

Cette structure représente la couleur que peuvent avoir les pions.

LIGNE

Ainsi nous utilisons la structure Couleur sous forme de pointeur dans la structure suivante :

```
typedef struct ligne{
Couleur *pion; //on allouera dynamiquement en fonction de la taille de la ligne que propose l'utilisateur
int nbrBonneCouleurBonEndroit;
int nbrBonneCouleurMauvaisEndroit;
}ligne;
```

Cette structure représente une ligne de proposition faite par le joueur. En effet, comme expliqué ci-dessus dans les règles, le joueur fait une proposition donc une ligne de pion de couleur que nous interprétons dans le pointeur pion. Ainsi après avoir fait sa proposition, l'adversaire donne le nombre de pion de la bonne couleur et au bon endroit et le nombre de pion de bonne couleur mais au mauvais endroit.

L'allocation de mémoire pour le pointeur pion se fait lorsque le joueur donne l'information du nombre de colonne qui souhaite mettre en jeu lors de la partie.

JOUEUR

Le joueur de son côté possède la structure suivante :

```
typedef struct joueur
{
   char nom[lengthName];
   mastermind proposition;
   int score;
} joueur;
```

Afin de stocker l'ensemble de la partie en cours, nous avons opté pour un pointer de ligne.

```
typedef ligne *mastermind;
```

Ainsi, vous remarquerez que le joueur possède un attribut proposition de type mastermind. Cet attribut permet de stocker l'ensemble de la partie en cours. A chaque fois que le joueur

remplit une ligne de couleur, celle-ci est ajoutée dans le tableau de pointer. L'allocation de la mémoire de ce tableau de pointeur qui est l'attribut proposition est faite dès que le joueur donne le nombre d'essais.

```
joueur->proposition = (ligne *)malloc(sizeof(ligne) * nombreEssais);
```

Nous avons fait le choix de faire l'allocation d'un seul coup, nous aurions pu cependant faire une allocation après chaque essai en utilisant realloc :

```
joueur->proposition = (ligne *)realloc(sizeof(ligne),numEssais);
```

Le résultat final aurait été quant à lui identique.

LES FONCTIONS

Afin de manipuler les structures que nous venons de détailler, nous avons dû implémenter un certain nombre de fonctions.

Les différentes fonctions que nous allons implémenter sont les suivantes :

```
/**
 * @brief Cette fonction permet au joueur de proposer une nouvelle ligne, après avoir proposé une ligne, celle-ci est analysée pour déterminer le nombre pions à la bonne place et de la bonne couleur et le nombre de pions mal placés mais de la bonne couleur. Une fois que la ligne est analysée, celle-ci est ajoutée au fichier de la partie.
 *
 * @param joueur
 * @param combinaison combinaison secrète que le joueur doit deviner
 * @param numeroEssai n° de l'essai en cours
 */
void playerInput (joueur *joueur, Couleur *combinaison, int numeroEssai);

/**
 * @brief Cette fonction permet de vérifier si le joueur a trouvé ou non la bonne combinaison. Elle retournera un booléen en fonction de la réussite ou non.
 *
 * @param targetLigne
 * @param input
 * @param lengthLine
 * @return true
 * @return false
 */
bool checkLineContent(Couleur *targetLigne, ligne *input);

/**
```

```
 * @brief Cette fonction a pour but de créer un fichier ayant pour nom le nom du joueur et de son adversaire si le fichier existe deja nous demandons au  joueur de donner un nom au fichier
 *
 * @param joueur
 *
*/
void createGameFile(joueur joueur);


/**
 * @brief Cette fonction a pour but d'ajouter au fichier crée préalablement la ligne que vient de rentrer le joueur ainsi que les nbrBonneCouleurBonEndroit et nbrBonneCouleurMauvaisEndroit.
 *
 * @param proposition ligne proposée par le joueur
 */
void addLigneToFile(ligne *proposition);

/**
 * @brief Cette fonction a pour objectif d'écrire la combinaison secrète sur la première ligne du fichier de sauvegarde
 *
 * @param combinaison
 */
void addFirstInfo(Couleur *combinaison, joueur *joueur);


/**
 * @brief Cette fonction a pour objectif de retrouver le nom du fichier dans le dossier Previous-Game.
 *
 */
void *findFileName();

/**
 * @brief Cette fonction a pour objectif de creer une nouvelle ligne objectif et d'afficher les anciennes decisions qui ont été prises par le joueur afin qu'il se retrouve dans le jeu.
 *
 * Cette fonction permet également d'instantier la combinaison secrète et les anciennes lignes proposées par le joueur.
 *
 *@param combinaison Combinaison secrete définit dans le main
 *
 *@param joueur joueur
 *
 */
```

```
void createGameFromFile(Couleur *combinaison, joueur *joueur);

/**
 * @brief Cette fonction permet de donner les valeurs aux paramètres du jeu en fonction de la ligne lu dans le fichier.
 *
 * @param line
 * @param joueur
 */
void setGameParameters(char *line, joueur *joueur)

/**
 * @brief Fill the secret combinaison from the line in the file
 *
 * @param line
 * @param combinaison
 */
void fillCombinaisonFromFile(char *line, Couleur *combinaison)



/**
 * @brief Fill the line from the line in the file (Colors, wrong and good answer)
 *
 * @param line
 * @param inputLine
 */
void fillInputLine(char *line, ligne *inputLine)

/**
 * @brief Cette function est utilisée pour supprimer un fichier à la demande de l'utilisateur.
 *
 * @return void permet de rappeller la fonction autant de fois que le joueur le souhaite
 */
void deleteFile()
```

LES VARIABLES EXTERNES

Afin de faciliter les échanges de variables entre les fonctions, nous avons utilisé des variables externes.

```
/// Number of column
```

```
extern int nombreColonne;
/// Number of trials
extern int nombreEssais;
/// Number of lines filled by the player
extern int playerLinesCount;
// Name of the game file
extern char filename[100];
```

Cela nous permet de pouvoir changer la valeur de ces variables facilement et de ne pas les passer systématiquement en paramètre.

MACROS ET CONSTANTES

MACROS

Dans notre projet, des tâches se répètent. Afin d'éviter une surcharge du code et d'apporter une meilleure lisibilité du code, nous avons implémenté des macros.

Convertir une chaine de caractère en entier :

```
#define TO_INT(x) atoi(x)
```

Faire un retour chariot :

```
#define RETURN printf("\n")
```

Afficher un message dans le terminal avec les informations sur le fichier, la ligne :

```
#define ASSERT(message) fprintf(stderr, __FILE__ ":%d: " message "\n", __LINE__)
```

Afficher un message avec une variable de type entier avec les informations sur le fichier, la ligne :

```
#define ASSERT_INT(message, data) fprintf(stderr, __FILE__ ":%d: " message " = %d\n", __LINE__, data)
```

Afficher un message avec une variable de type chaine de caractères avec les informations sur le fichier, la ligne :

```
#define ASSERT_DATA(message, data) fprintf(stderr, __FILE__ ":%d: " message " = %s\n", __LINE__, data)
```

CONSTANTES

Longueur max du nom du joueur :

```
#define LENGTH_NAME 20
```

Chaine de caractère représentant la commande pour nettoyer le contenu du terminal :

```
#define CLEAR "cls||clear"
```

Les || représentent la condition « ou », cela permet de lancer le programme aussi bien sur ordinateur avec un OS Windows ou Linux/MacOS.

Nom du dossier dans lequel se trouve les fichiers de toutes les parties :

```
#define FOLDER_NAME "Previous-Game/"
```

COMPILATION

Afin de compiler le code, nous avons mis en place un MakeFile. Comme nous l'avons vu lors de ce semestre, le makefile est un fichier qui permet de compiler le code source et de générer un exécutable. Pour se faire il faut utiliser le logiciel Make.

```
Projet-LP25$ make
```

L'outil de compilation que nous utilisé est GCC (GNU compiler collection).

Une autre méthode pour compiler le programme peut être de faire la compilation directement dans le terminal avec gcc pour cela il faut taper la commande suivante :

```
Projet-LP25$ gcc -g src/*.c -o Mastermind
```

L'option -g permet de générer les informations symboliques de débogage. Quant à l'option -o, celle-ci fixe le nom du fichier objet généré lors de la compilation d'un fichier source.

L'inconvénient de compiler directement dans le terminal est que les fichiers objets (.o) ne sont pas supprimer automatiquement. En effet, nous avons ajouté un phoney dans le MakeFile.txt ce qui permet de supprimer automatiquement les fichiers .o après la compilation. De plus la personnalisation de la compilation est plus simple en utilisant Make car nous pouvons ajouter les flags ou autre facilement, chose compliqué à faire lorsque nous utilisons gcc directement en ligne de commande car dans un projet de grande, la ligne de commande peut vite devenir compliquée à écrire. C'est donc pour cela que nous avons opté pour un MakeFile.

SAUVEGARDE DES PARTIES

Afin de sauvegarder les parties en cours, nous avons créé un dossier Previous-Game/ dans lequel toutes les anciennes parties sont stockées.

Chaque fichier à la structure suivante :

Example.csv

NomJoueur;nombreColonne;nombreEssais;
CombinaisonSecrete
Ligne0
Ligne1
Ligne2

Voici à quoi ressemble un fichier complété

Paul;4;6;
5;2;5;4;
2;5;1;5;0;3;
5;5;4;2;1;3;
5;5;1;2;1;2;

Concernant les lignes celle-ci sont composées pour les n nombre de colonnes de l'index de la couleur dans l'enum Couleur choisit par le joueur lors de la partie qu'il avait joué et les deux dernières valeurs sont nombre de couleurs au bon endroit et le nombre de bonnes couleurs mais mal placées. Ainsi grâce à cette structure de fichier, nous pouvons recréer une partie. Afin de recréer une partie, nous devons convertir les index en couleur pour cela nous avons implémenter la fonction `Couleur getColorByIndex(int index).`

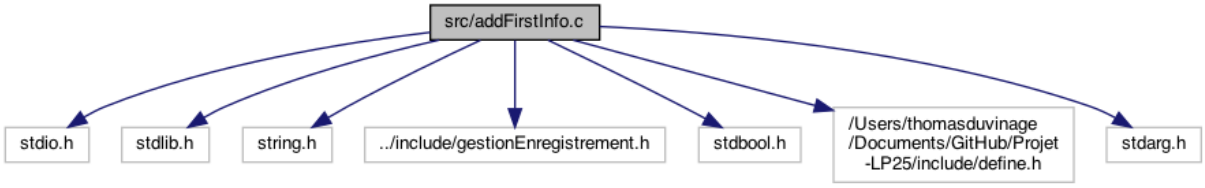
Toute partie qu'un joueur commence est sauvegardée dans un fichier, lorsque le joueur souhaite quitter la partie en cours en donnant comme position le nombre de colonne + 1, un message lui demande s'il souhaite conserver la partie en cours.

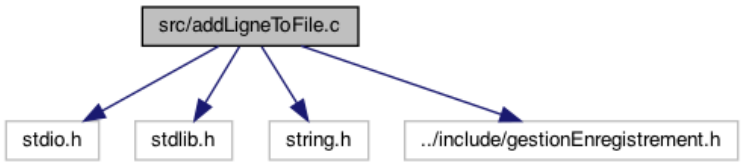
Cependant le joueur peut dire oui mais ne jamais reprendre la partie c'est donc pour cela que nous avons donné la possibilité au joueur de supprimer des anciennes parties comme bon lui semble.

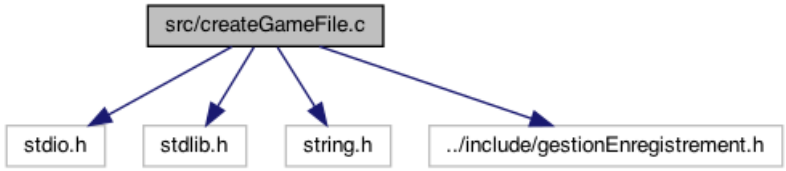
L'inconvénient de cette méthode est que n'importe quel joueur peut supprimer la partie de quelqu'un d'autre. Afin d'éviter cela, nous aurions pu créer un dossier pour chaque joueur, et stocker les parties de ce joueur dans celui-ci. Ainsi à chaque fois que le joueur se connecte il n'a accès uniquement à son dossier.

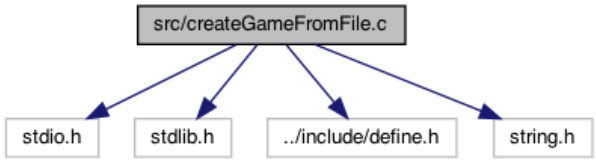
ANNEXES

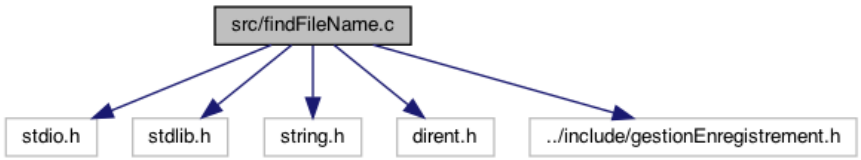
Dépendances de tous les fichiers sources (graphiques réalisés via Doxygen):

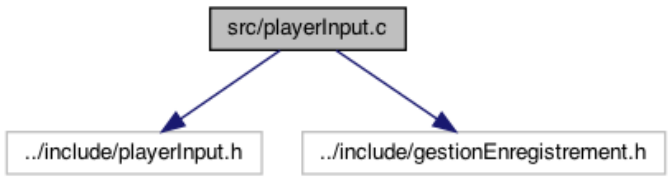


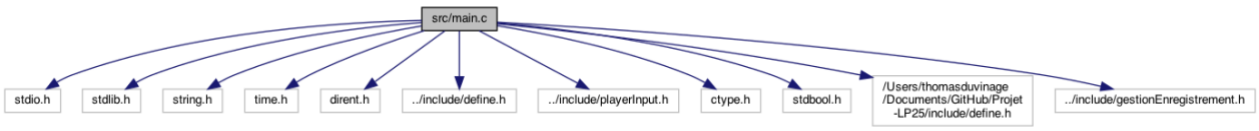

src/addLigneToFile.c
stdio.h
stdlib.h
string.h
../include/gestionEnregistrement.h


src/createGameFile.c
stdio.h
stdlib.h
string.h
../include/gestionEnregistrement.h


src/createGameFromFile.c
stdio.h
stdlib.h
../include/define.h
string.h


src/findFileName.c
stdio.h
stdlib.h
string.h
dirent.h
../include/gestionEnregistrement.h


src/playerInput.c
../include/playerInput.h
../include/gestionEnregistrement.h


src/main.c
stdio.h
stdlib.h
string.h
time.h
dirent.h
../include/define.h
../include/playerInput.h
ctype.h
stdbool.h
/Users/thomasduvinage
/Documents/GitHub/Projet
-LP25/include/define.h
../include/gestionEnregistrement.h