
Projet

2048

TRINH Donovan

M.P.I – L1 – S1

Université Paris-Sud - Informatique

2017

Professeurs :

THIERY Nicolas, GAY Joël

INTRODUCTION

J'ai choisi de faire **le Projet du 2048**. Le but va être d'implanter toutes les fonctions nécessaires à la **création d'un jeu 2048**.

Celui-ci sera premièrement affiché en console, et le joueur interagira aussi par l'intermédiaire de cette console. Le jeu devra respecter les règles basiques du 2048, afficher le plateau, et prendre en compte les choix de l'utilisateur. Ce sont les fonctions basiques du jeu, une fois fonctionnel, il pourra évoluer. Et ainsi on pourra tenter d'y intégrer une interface graphique (des couleurs dans un premier temps à ainsi que des commandes simplifiées (avec les flèches) et d'autres fonctionnalités (comme un score).

Plutôt que de lister toutes les fonctionnalités implantées en première partie puis faire une seconde partie sur les difficultés rencontrées, j'ai préféré opter pour un rapport sous forme **d'un journal de bord** qui réunit ces deux parties. On peut ainsi suivre tout l'avancement du projet par ordre chronologique, voir chaque problème rencontré, et la façon dont il a été résolu.

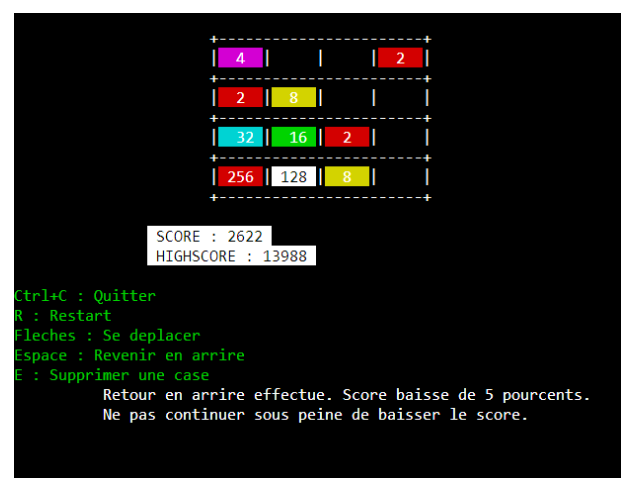
EDIT : 6 Décembre

Le jeu est à présent terminé. Il possède toutes les fonctions basiques suivantes :

Déplacement avec les flèches, jeu en couleur qui s'actualise à chaque coup, gestion des mouvements impossibles ou touches invalides, blocage du jeu lorsque celui-ci est gagné ou perdu, possibilité de relancer une partie avec R, affichage du score.

Il possède les améliorations suivantes :

Enregistre et affiche le Highscore, celui-ci se réactualise à chaque coup s'il est battu au cours d'une partie. Possibilité d'un retour en arrière avec la touche Espace, mais entraîne une perte de score de 5%. Augmente ou Diminue une case aléatoire du jeu avec des probabilités respectives de 2% et 4% à chaque coup. Règles précises page 43.



AGENDA DES EVENEMENTS

11 Novembre :

Téléchargement de CodeBlocks pour coder en C++. Création du journal de bord de rapport.
Début de la programmation, les premières fonctions sont **imaginées**.
Premiers **problèmes rencontrés**, lié à la compilation via CodeBlocks.

13 Novembre :

Les fonctions d'affichages sont **terminées**.
Création des premiers tests.

14 Novembre :

Les fonctions de déplacements sont **terminées**.

15 Novembre :

Les premières fonctions liées à l'aléatoire sont **créées**.
Début de la programmation en compilation séparée. Installation de Dreamweaver pour coder, et Bash pour le terminale Linux.

16 au 19 Novembre :

Fichiers annexes de test **terminés**.
Le jeu basique (sans extensions) fonctionne.

20 Novembre :

Tous les bugs sont **corrigés**. Le jeu est prêt à recevoir des extensions.

22 Novembre :

Implantation du **score** et corrections de quelques bugs.

23 au 30 Novembre :

Etude de la bibliothèque **NCurses**, sans codage. Uniquement l'apprentissage des fonctions existantes.
Création de la Documentation annexe sur NCurses.

31 Novembre et 1^{er} Décembre :

Implantation du nouvel affichage en couleur, actualisation automatique du plateau de jeu, utilisation des flèches au lieu d'entrer une commande. Amélioration de l'affichage (plus clair).

2 et 3 Décembre :

Création du **Makefile** et de la première sauvegarde (commit) sur **GIT**. Création des types Struct.

4 et 5 Décembre :

Lutte et **résolution d'erreurs** majeures, en vue d'amélioration du code.

6 Décembre :

Perfection du code. Ajout de **HighScore** via un fichier externe. Ajout du **type Struct**, qui va faciliter la suite. Ajout du début de ma variante, soit de la fonction **baisseOuAug**.

7 et 8 Décembre :

Ajout de la possibilité de **Suppression**. Changements des Touches. La version finale est **terminée**.

JOURNAL DE BORD

Journal tenu jusqu'à la fin de la création du 2048 de base sans extensions. La suite ne sera pas datée.

11 Nov :

J'ai d'abord imaginé afficher un plateau de jeu à base d'un unique tableau 2D comportant soit des int pour la valeur de la case, soit une chaîne de caractère pour les « * ». Il comporterait donc alternativement des int puis des string. Abandon de ce système car il nécessite un tableau 2D (9x9) contenant une template class (pour avoir à la fois des int et des string). Bien trop compliqué, et peu pratique.

On va plutôt opter pour un **tableau 2D d'int (4x4)** qui contient chaque valeur du plateau de jeu. Toutes les étoiles*** seront construites par des cout. L'affiche se fera pas un enchaînement de cout, de *, et de t[i][j] pour les int .

J'ai voulu créer un plateau initialisé avec des 0 ou avec certaines valeurs, mais lors de l'allocation du tableau, mon compiler n'accepte pas le code sous cette forme :

```
plateau = { {0,0,0,0}, {0,0,0,0}, {0,0,0,0}, {0,0,0,0} };
```

Il me fait une erreur (EDIT : Erreur résolue plus loin) :

Warning : extended initializer lists only available with -std=c++11 or -std=gnu++11

Après une rapide recherche, on peut envisager l'utilisation de **setw()** pour définir la taille d'une case d'un tableau lorsqu'on l'écrit avec un cout. Cette fonction nécessite d'inclure une autre bibliothèque **#include <iomanip>**.

12 nov :

J'ai fait plusieurs recherches sur les bibliothèques, et je vais exploiter **<iomanip>** pour l'affichage de mon plateau.

La commande **setw(int)** allié à un **cout** permet de créer une longueur précise pour notre flux. Par exemple, `cout << setw(5) << "+"` va créer " +" avec 4 espaces vides et un +. On peut aussi remplacer les espaces (par défauts) par un autre signe, comme "*" avec une commande supplémentaire.

C'est donc ce qu'on va utiliser pour les lignes de séparations, composées de 25 étoiles *, avec la commande **setfill('*')**. Ce qui nous donne :

```
Cout << setfill('*') << setw(25) << endl;      //car il faut 25 '*'
```

Ce qui nous donnera en sortie :

```
*****
```

On va également utiliser ces deux fonctions (`setw` et `setw`) pour emplir nos autres cases, comme celle du 4 :

```
* 4 * 8 * ...
```

Voici la structure d'une **case** :

- composée de 5 caractères
- contenant **au centre** la valeur de la case (*cf paragraphe suivant*)
- séparées entre elles par un *
- la ligne débute également par un *

Par exemple :

```
*****
* 4 * 128 * 16 * 1024*
*****
```

On remarque que chaque valeur de plus d'1 chiffre (soit dans l'exemple 16, 128, et 1024) n'a qu'un espace à gauche puis un alignement à gauche alors que les valeurs d'1 chiffre (2, 4 ou 8) auront 2 espaces puis un alignement à gauche. Une valeur de 0 entraîne aussi un espace vide. On peut donc **séparer ces trois cas** de cette façon :

```
if (valeurDeCase == 0) { cout << setw(5);} else {
if (valeurDeCase < 10) { cout << " " << valeurDeCase << " " ;}
else { cout << " " << left << setw(4) << ValeurDeCase ;} }
```

On peut ensuite construire notre tableau en paramétrant une ligne avec des boucles `for` et en implantant la fonction précédente :

```
Void afficheCase(int valeurDeCase) {
    if (valeurDeCase == 0) { cout << setw(5);} else {
        if (valeurDeCase < 10) { cout << " " << valeurDeCase << " " ;}
        else { cout << " " << left << setw(4) << ValeurDeCase ;} }
    cout << "*" ; //affiche une étoile après la case
}

Void afficheSeparation() {
    cout << setfill('*') << setw(25) << endl;
}

Void afficheLigne (int numLigne, vector< vector<int> > t) {
    Cout << "*" ;
    For (int i=0 ; i<4 ; i++) {
        afficheCase(t[numLigne][i];
    }
}

Void affichePlateau (vector< vector<int> > t) {
    afficheSeparation();
    afficheLigne(0, t);
    afficheSeparation();
    afficheLigne(1, t);
    afficheSeparation();
    afficheLigne(2, t);
    afficheSeparation();
    afficheLigne(3, t);
    afficheSeparation();
}
```

L'idée est là, il faudra ensuite adapter les noms de variables au projet et écrire la documentation de chaque fonction. La fonction d'affichage est maintenant créée, il faudra faire des tests.

13 nov :

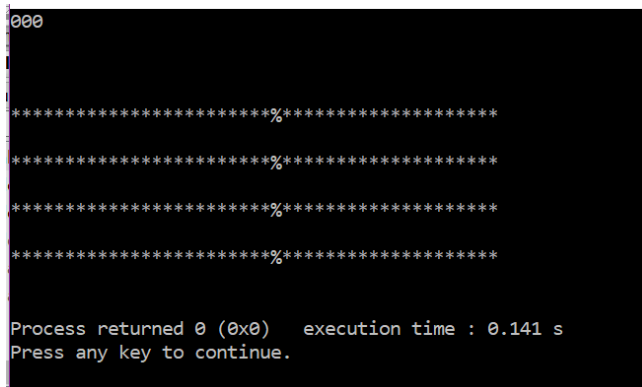
Comme la commande précédente (cf 11 novembre) ne fonctionnait pas pour initialiser le tableau en une ligne avec des valeurs souhaitées, on va initialiser les cases une à une pour commencer des premiers aperçus/test des fonctions d'affichage :

```
plateau [0][0]=1; plateau [1][2]=8; plateau [2][3]=16;  
plateau [1][0]=128; plateau [3][2]=1024; plateau [3][3]=1;
```

Après avoir commencé à implanter les fonctions, je m'aperçois de plusieurs problèmes d'affichage. Le plateau n'affiche que des étoiles, et il semble y en avoir plus que les 25 prévus.

Je remplace la première étoile des lignes contenant des cases d'entiers par un % pour voir où elles commencent. J'affiche également certaines cases du tableau (t[0][0] ; t[1][2] ; t[3][3]) en début de console pour vérifier que celles-ci existent et valent 0.

Cette dernière action m'a permis de voir que je n'avais pas un tableau de 4x4 mais de 4x3, t[3][3] n'existait pas, il manquait une ligne. Une fois l'erreur rectifiée. Voici ce que j'obtiens :



```
000  
  
*****%*****  
*****%*****  
*****%*****  
*****%*****  
  
Process returned 0 (0x0) execution time : 0.141 s  
Press any key to continue.
```

Il reste donc des erreurs à rectifier. Il y a sûrement **des problèmes dans les retours aux lignes, et dans l'affichage des cases** qui ne sont visiblement remplies que par des * au lieu de leurs valeurs.

Rectification de l'erreur :

Il semble que le `cout` n'accepte pas de retour à la ligne après un `setw(25)` s'il n'y a pas de caractère contenu dedans. J'ai donc rajouté comme dernier caractère une étoile. J'ai également raccourci le nombre d'étoiles qui était incorrecte, à 21. Ce qui nous donne :

```
cout << setfill('*') << setw(21) << "*" << endl;
```

L'affichage d'un tableau vide peut maintenant se faire :

```
000
*****
%      *      *      *      *
*****
%      *      *      *      *
*****
%      *      *      *      *
*****
%      *      *      *      *
*****
```

On souhaite à présent tester notre affichage avec un tableau contenant des valeurs d'un seul chiffre en déclarant le tableau ainsi :

```
plateau = { {1,2,3,4}, {2,3,4,5}, {3,4,5,6}, {4,5,6,7} };
```

Or le code ci-dessus ne fonctionne toujours pas, on obtient toujours la même erreur que le 11 novembre (page 4).

On va donc le faire manuellement :

```
plateau [0][0]=1; plateau [1][2]=8; plateau [2][3]=16;
plateau [1][0]=128; plateau [3][2]=1024; plateau [3][3]=1;
```

Je m'aperçois alors que certaines lignes ne sont plus de la bonne taille. En fait les lignes d'étoiles `*` doivent bien contenir 25 étoiles, ce sont mes cases vides qui ne faisaient que 4 espaces vides au lieu de 5. Après avoir corrigé cela, voici le résultat du test :

```
*****
% 1  *      *      *      *
*****
% 128 *      *      8      *
*****
%      *      *      * 16  *
*****
%      *      * 1024* 1    *
*****
```

Les implantations des fonctions d'affichage sont donc terminées. Il ne reste plus qu'à créer un second fichier `modele.h` dans lequel je vais les déclarer. Je m'occuperai de ça après avoir fait le TP qui porte sur cette gestion des fonctions via plusieurs fichiers. En attendant je vais commencer à coder d'autres tests, comme la vérification du plateau gagnant, ou la vérification du plateau perdant (rempli).

Tests de possibilités de mouvements :

Le premier test écrit est celui du **plateau gagnant**. Il suffit de deux boucles for l'une dans l'autre pour scanner le tableau à la recherche d'une valeur `plateau[i][j] >= 2048`.

Pour **savoir si les déplacements sont possibles**, on va diviser en 4 tests, chacun testant un côté. Voici l'idée du fonctionnement, pour un déplacement vers le haut, le déplacement ne sera possible que si une de ces deux conditions sont vérifiées :

- Il existe au moins une case dont celle du dessus est vide (ainsi elle pourra monter).
- Il existe au moins une case non nulle dont celle du dessus est égal à cette case (ainsi elle pourra fusionner avec, fonction à coder plus tard).

On va donc balayer tout le plateau (sauf première ligne) en vérifiant pour chaque case si celle du dessus est vide, ou si elle est égal à la case visée (et non nulle). Voilà comment cela se traduit en code :

```
bool movmtPossibleHaut(Matrice plateau) {
    string possible = "non";
    for (int i=1 ; i<4 ; i++) { //i=1 car la première ligne ne peut pas être
testée
        for (int j=0 ; j<4 ; j++) {
            if (plateau[i][j] == plateau[i-1][j]) {
                possible = "oui" ;
            } else {
                if (plateau[i][j] != 0 and plateau[i-1][j]) {
                    possible = "oui";
                }
            }
        }
    }
    if (possible=="oui") {
        return true;
    } else {
        return false ;
    }
}
```

J'ai donc introduit une variable string "possible" afin de faciliter le test. Elle est initialisée à "non", on part du principe que le test est faux, donc que la grille est bloquée. Et Dès que le balayage rencontre une case où il est possible de faire un mouvement, alors il affecte "oui" à la variable. En fin, si `possible=="oui"`, alors la fonction renvoie True.

La fonction fonctionne, mais elle n'est pas optimisée. On pourrait rajouter une boucle while, afin que la fonction ne continue que tant que la variable n'est pas égale à "oui". Car pour l'instant, même lorsqu'elle détecte que le test est possible, elle va continuer de scanner tout le plateau.

Rectification :

Au lieu de 'possible = "oui" ' on va instaurer un 'return true' et mettre un 'return false' à la fin de la double boucle for. Cette nouvelle fonction est beaucoup plus simple et épuré puisqu'elle s'arrête si le mouvement est possible au lieu de continuer inutilement. Voici la version finale à priori, avec sa documentation, de la fonction **Test de mouvement possible** :

```
/** Test s'il est possible de faire ou mouvement vers le haut ou si le jeu est bloqué
 * Test 1/4 (4 côtés)
 * @param le plateau de jeu
 * @return True si le mouvement possible, Else sinon
 */
bool movmtPossibleHaut(Matrice plateau) {
    for (int i=1 ; i<4 ; i++) { //i=1 car la première ligne ne peut pas être testée
        for (int j=0 ; j<4 ; j++) {
            if (plateau[i][j] == plateau[i-1][j]) {
                return true;
            } else {
                if (plateau[i][j] != 0 and plateau[i-1][j]) {
                    return true; } else {} //le else n'a pas d'utilité
            }
        }
    }
    return false ;
}
```

J'ai mis un `else{}` vide, celui-ci n'a pas d'utilité dans le code. Il me sert simplement à mieux visionner ma fonction entre les boucles For et les If qui s'enchaînent. Maintenant, pour créer la fonction `estTermine` qui test si tous les mouvements sont bloqués, il suffit de vérifier qu'il y a au moins un mouvement de possible. Il faudra créer toute une série de plateau 4x4 afin de tester chaque fonction.

14 nov :

J'ai fini d'implanter mes 4 fonctions qui déplacent les cases. Mais ces fonctions ne prennent pas en compte la fusion. Voici le code celle du déplacement vers le haut :

```
/** Fait remonter toutes les cases vers le haut de façon qu'aucune valeur ne possède de vide au dessus d'elle
 * @param le plateau de jeu
 * @return le plateau avec un mouvement partiel (sans fusion des nombres identiques)
 */
Matrice deplacementPartielHaut(Matrice plateau) {
    for (int k=0 ; k<3 ; k++) { // L'opération se répète 3 fois pour être sûr qu'une case du bas monte tout en haut
        for (int i=0 ; i<3 ; i++) { // Sur les 3 premières lignes uniquement
            for (int j=0 ; j<4 ; j++) {
                if(plateau[i][j] == 0) { // Si case vide, elle est remplacée par celle du dessous
                    plateau[i][j] = plateau[i+1][j];
                    plateau[i+1][j] = 0;
                } else { }
            }
        }
    }
    return plateau;
}
```

J'appelle cette fonction (cf Page Précédente) **deplacementPartiel** car ce n'est pas un déplacement complet comme celui souhaité par le joueur lors d'un "coup", il ne prend pas en compte la fusion de 2 cases de valeur identiques, mais seulement des déplacements. Elle fonctionne ainsi :

Pour un déplacement vers le haut, **seules les 3 premières lignes seront balayées** (soit de $i=0$ à $i=2$). Pour chaque case (`plateau[i][j]`), si celle ci est vide, alors elle prend la valeur de la case du dessous, et la case du dessous prend la valeur de 0.

On a donc "remonté" la case d'1 cran. Il faudra faire **une version pour chaque côté, en veillant à bien changer les boucles for**. Par exemple, pour les déplacements vers la droite, il faut partir de la dernière colonne, et tout balayer de façon à finir par la deuxième, sinon les déplacements ne sont pas complets.

Lors de ce processus, chaque case va être remontée d'un cran. Mais si jamais une case se trouve tout en bas, il aura besoin de remonter **3 fois pour se retrouver tout en haut**. On met donc le tout dans une troisième boucle for afin de répéter l'opération 3 fois.

On peut maintenant s'attaquer à la fonction de **fusion de 2 cases** lors du déplacement pour venir compléter la précédente. Avec ces deux fonctions (déplacements partiel + fusion) ainsi que le test (`mouvmtPossible?`) , on aura enfin tout ce qu'il faut pour créer la commande de déplacement "mouvementHaut" qui sera actionné par le joueur.

Le principe de cette fonction est de, lors d'un mouvement vers le haut, faire **fusionner deux cases qui auraient la même valeur et qui seraient l'une au-dessus de l'autre**. Ainsi nous allons balayer tout le tableau comme précédemment à la recherche de deux cases répondants aux critères de fusion. Ce qui nous donne cette fonction :

```
/** Fusionne des cases identiques par un mouvement vers le haut
 * @param le plateau de jeu
 * @return le plateau avec sa fusion
 */
Matrice fusionHaut(Matrice plateau) {
    for (int i=0 ; i<3 ; i++) { // Sur les 3 premières lignes uniquement
        for (int j=0 ; j<4 ; j++) {
            if(plateau[i][j] == plateau[i+1][j]) { // Si case identique à
celle du dessous, fusion des deux
                plateau[i][j] *= 2;
                plateau[i+1][j] = 0;
            } else { }
        }
    }
    return plateau;
}
```

Suite à cette fonction, des **espaces vides peuvent apparaître**, et il faut alors remonter les cases situées sous cet espace. Nous pourrions éventuellement réutiliser la fonction de mouvementPartiel, même si elle est **assez lourde** puisqu'elle scan 3 fois tout le plateau. Mais il serait aussi lourd de recréer une fonction qui ne scan le plateau qu'une fois, ou de l'incruster dans la fonction de fusion. Nous allons donc opter pour réutiliser la fonction de Déplacement partiel quitte à balayer inutilement le plateau de jeu.

Si un soucis de **rapidité** se fait ressentir (le jeu répond lentement), ceci sera une première piste à exploiter pour alléger les calculs nécessaires et donc entraîner un gain de rapidité de réaction.

On peut maintenant lier nos 2 fonctions ainsi que le test pour faire notre fonction de déplacement.

Si le mouvement est possible, on déplace les cases, fusionne ce qui doit fusionner, et on remonte les cases. Nous introduisons donc une variable locale en plus (launchscan) :

```
Matrice fusionHaut(Matrice plateau) {
    int launchscan = 0;
    for (int i=0 ; i<3 ; i++) { // Sur les 3 premières lignes uniquement
        for (int j=0 ; j<4 ; j++) {
            if(plateau[i][j] == plateau[i+1][j]) { // Si case identique à
celle du dessous, fusion des deux
                plateau[i][j] *= 2;
                plateau[i+1][j] = 0;
                launchscan = 1;
            } else { }
        }
    }
    if (launchscan==1) { deplacementPartielHaut(plateau) ; } // remonte les
cases si nécessaires
    else {} // else sans utilité, simplement pour plus de clareté
    return plateau;
}
```







Nous avons donc maintenant notre **fonction finale de déplacement**, celle qui sera actionnée par le joueur :

```
/** Exécute un mouvement vers le haut à la demande du joueur
 * @param le plateau de jeu
 * @return ce même plateau avec le mouvement exécuté
 */
Matrice deplacementHaut(Matrice plateau) {
    if ( mouvementPossibleHaut(plateau) ) {
        plateau = deplacementPartielHaut(plateau);
        plateau = fusionHaut(plateau) ;
    }
    return plateau;
}
```

15 Nov :

Le TP sur **la compilation séparée** a été fait en classe, je suis donc maintenant en mesure de me créer un fichier .h afin d'y mettre de l'ordre dans mes fonctions. J'ai donc créé un nouveau dossier, où j'y ai rangé toutes mes fonctions en 6 fichiers (3 paires de .h et .cpp) comme voici : Les fichiers .h contiennent la liste des fonctions (par thème) mais aussi leurs documentations.

Nom

-  affichages.cpp
-  affichages.h
-  déplacements.cpp
-  déplacements.h
-  verifications.cpp
-  verifications.h

- Affichages traite tout ce qui est nécessaire pour afficher le tableau. Au final, seul la fonction "void affichagePlateau(Matrice plateau)" va être appelée en dehors.
- Dans vérifications, nous avons les tests "estBloqué" et "estGagné", mais aussi des tests qui vérifient pour chaque direction si le mouvement est possible.
- Dans déplacements, il y a toutes les fonctions nécessaires à la réponse à l'utilisateur. Les fonctions finales à appeler sont "deplacementHaut" où Haut peut être remplacé par Droite, Bas ou Gauche.

Voilà un exemple de fichier, il s'agit du contenu du verifications.h :

```
1  /** Verifie certains états du jeu ou certaines possibilités
2   * Le mouvement est-il possible ? , Le jeu est-il bloqué ? Ou Gagnant ?
3   */
4
5
6
7  /** Test s'il est possible de faire un mouvement vers le [HAUT] ou si le jeu est bloqué
8   * @param le plateau de jeu
9   * @return True si le mouvement possible, Else sinon
10  */
11  bool movmtPossibleHaut Matrice plateau;
12  bool movmtPossibleDroite Matrice plateau;
13  bool movmtPossibleBas Matrice plateau;
14  bool movmtPossibleGauche Matrice plateau;
15
16
17  /** Test si un mouvement possible ou si le jeu est bloqué
18   * @param le plateau de jeu
19   * @return True si aucun mouvement dans une des 4 directions n'est possible, Else sinon
20  */
21  bool estBloque Matrice plateau;
22
23
24  /** Test si le jeu est gagné, soit qui contient une valeur d'au moins 2048
25   * @param le plateau de jeu
26   * @return True si le jeu est gagné, Else sinon
27  */
28  bool estGagne Matrice plateau;
29
30
31
```

Screenshot pris sous le logiciel de codage CodeBlocks, adapté aux projets en C++.

Par la suite, j'utiliserai DreamWeaver (un logiciel de la série d'Adobe), plus adapté pour du codage Web, mais dont l'interface est plus agréable à mon goût.

Je me demande si le fait d'utiliser le type Matrice dans les fichiers .h où aucune bibliothèque/références n'est faite ne va pas entraîner des erreurs.

Je me pose des questions car j'utilise dans certains fichiers des fonctions qui font appels à des fonctions déclarées dans plusieurs fichiers .h. Je m'attends à rencontrer des erreurs lors des compilations.

J'ai rajouté une fonction test comme suivi, afin de vérifier si le plateau est plein :

```
bool grillePleine(Matrice plateau){
    for (int i=0 ; i<4 ; i++) {
        for (j=0 ; j<4 ; j++) {
            if (plateau[i][j] > 0) {}
            else { return false ; } // false si une case est vide
        }
    }
    return true;
}
```

Cette fonction de test est nécessaire pour créer la fonction qui va faire apparaître aléatoirement le prochain numéro sur une case vide.

Conception de la fonction de Random Spawn

Voici les caractéristiques que cette fonction (ou ensemble de fonctions) doit respecter :

- Elle vérifie si le plateau possède de la place pour faire Spawner (apparaître) un nombre. S'il est plein, le jeu est terminé
- elle fera apparaître les nombres 2 ou 4 avec des probabilités respectives de 9/10 et 1/10
- ce nombre devra apparaître sur une case aléatoire vide du plateau
- chaque case vide doit avoir la même probabilité de recevoir un spawn

On va premièrement s'intéresser à la **fonction aléatoire**.

Afin que chaque case vide ait la même probabilité de spawner le nombre, on va premièrement **compter le nombre de case vide**. On fera ensuite une **boucle while** qui balayera ces cases vides en proposant une probabilité de spawn de $1 / [\text{Nbr de Case Vide}]$ pour chaque case. La boucle while sert à passer d'une case vide à l'autre tant que le spawn n'a pas eu lieu.

Une fois la case choisie par cette première fonction, il en faudra une seconde qui décidera si cette case est un 2 ou un 4.

Pour le fonctionnement de la création de notre nombre aléatoire, pour une probabilité de $1/N$. on va **générer un nombre aléatoire entre 1 et N**. Si ce nombre est 1, alors l'évènement est validé. Par exemple :

Pour notre nombre 4 qui apparaît 1 fois sur 10, on génère un nombre entre 1 et 10 (inclus). Si ce nombre est 1, alors ce sera 4 qui apparaît, sinon, ce sera un 2.

Pour faire appel à ces fonctions, nous auront besoin de bibliothèques supplémentaires qui sont **<cstdlib>** et **<ctime>**.

On introduit chaque fonction qui utilise de l'aléatoire par le code suivant, ainsi il permet de ne pas avoir sans cesse la même suite de nombre aléatoire :

```
srand( time(0) )
```

On va ensuite utilisant un assemblage de plusieurs fonctions. `rand()` sert à choisir un entier aléatoire entre 1 et un très grand nombre. Ce très grand nombre correspond à "RAND_MAX". Ainsi `rand() / (double)RAND_MAX` nous renvoie un nombre floatant (double) compris entre 0 et 1.

On va multiplier ce nombre par N. Nous avons donc notre nombre floatant compris entre 0 et N. La probabilité que ce nombre soit inférieur à 1 est égal à la $1/N$.

Exemple :

Je veux savoir si un évènement de probabilité $1/4$ se produit. Je génère un nombre floatant entre 0 et 1 grâce à ma commande ci-dessus. Je le multiplie ensuite par 4. Mon nombre se trouve donc soit entre 0 et 1, 1 et 2, 2 et 3, ou 3 et 4. Il a donc 4 intervalles de mêmes tailles où il peut se situer, il a donc $1/4$ de chance de se trouver dans chacun des 4.

Pour simplifier les choses, on ne prendra que des nombres entiers à la place de N, mais qui seront de type floatant. Ainsi notre **fonction sera de type booléenne**. On peut donc écrire notre fonction qui nous **dit si un évènement de probabilité $1/n$ apparaît** :

```
bool eventProba1surN(float n) {
    srand( time(0) ); //assure que le nombre soit bien prit au hasard
    float x = rand() / (double)RAND_MAX ;
    float x *= n ;
    if (x <= 1) {
        return true;
    } else {
        return false; }
}
```

Nous allons donc avoir besoin d'une fonction qui **compte le nombre de cases vides** pour tester sur chaque case la probabilité de spawn. Ne pas oublier que cette fonction renvoie un entier mais de type Float pour être compatible avec la fonction ci-dessus. On a donc :

```
float nbrCasesVides(Matrice plateau){
    float compteur=0;
    for (int i=0 ; i<4 ; i++) {
        for (int j=0 ; j<4 ; j++) {
            if ( plateau[i][j] == 0) {
                compteur++;
            }
        }
    }
    return compteur;
}
```

On peut donc maintenant faire une fonction qui nous dit si le **nombre à sortir est un 2 ou 4**, en utilisant la fonction pour savoir si l'évènement de probabilité $1/10$ se produit :

```
int deuxOuQuatre(){
    if ( eventProba1surN(10) ) {
        return 4;
    } else {
        return 2 ;}
}
```

Afin de finaliser notre partie sur l'aléatoire, il nous faut une fonction qui va décider où placer le nouveau nombre, et qui va retourner le plateau de jeu avec ce nombre.

Il faut donc compter le nombre de cases vides, pour chaque case vide, lancer le programme d'évènement. Celui-ci inclus dans une boucle while, il faut ensuite lancer le dernier programme qui décide si c'est un 2 ou un 4 qui doit être instauré, puis l'installer à l'emplacement décidé par la fonction. J'appelle cette fonction **randomspawn** qui signifie "Apparition/naissance aléatoire" en anglais :

Version temporaire :

```
Matrice randomspawn(Matrice plateau){
    int confirmationSpawn=0 ;
    int nbrQuiSpawn = deuxOuQuatre() ; // choisi entre 2 et 4
    float casesVides = nbrCasesVides(plateau) ;
    while ( confirmationSpawn == 0) {
        for (int i=0 ; i<4 ; i++) {
            for (int j=0 ; j<4 ; j++) {
                if ( plateau[i][j] == 0) { //Si la case est vide
                    if ( eventProba1surN(casesVides) ) {
                        if (confirmationSpawn == 0) {
                            plateau[i][j] = nbrQuiSpawn ;
                            confirmationSpawn = 1 ;
                        }
                    }
                }
            }
        }
    }
    return plateau;
}
```

(Voir le fichier randomspawn.cpp pour voir la version avec les commentaires)

Après avoir codé la fonction je me rends compte qu'il n'y a pas d'utilité à mettre une variable de confirmation. Car le rôle de celle-ci est d'empêcher la fonction de continuer à scanner les cases vides. Mais on peut stopper la fonction par un simple return. On va donc alléger la fonction en retirant cette variable et en remplaçant correctement notre return.

La fonction nécessite quand même une **boucle while** afin qu'elle redémarre un scan du plateau au cas où elle n'ait pas placé le nombre aléatoire lors du premier scan. Le while contiendra une condition infinie pour faire tourner la fonction jusqu'à ce qu'elle return le plateau.

Fonction **randomspawn** :

```
Matrice randomspawn(Matrice plateau){
    int nbrQuiSpawn = deuxOuQuatre() ; // choisi entre 2 et 4
    float casesVides = nbrCasesVides(plateau) ;
    int confirmation=0;
    while (confirmation == 0) {
        for (int i=0 ; i<4 ; i++) {          // Balaye le plateau à la recherche
de cases vides
            for (int j=0 ; j<4 ; j++) {
                if ( plateau[i][j] == 0) { //Si la case est vide
                    if ( eventProba1surN(casesVides) ) { // Si l'event
est validé
                        plateau[i][j] = nbrQuiSpawn ; // On assigne le nombre
à la case
                        return plateau;
                    confirmation++; //inutile, juste au cas où, pour
éviter un problème avec une boucle infinie
                }
            }
        }
    }
}
```

Toutes les fonctionnalités principales du jeu sont maintenant codées, il ne reste plus qu'à les tester. Après ces tests, il faudra tout assembler et coder la partie interaction avec le joueur (demande du mouvement souhaité, réaction du jeu, ...).

COMPILATION SEPARÉE

Dans la prochaine partie, nous aurons besoin de faire de la **compilation séparée**, en dehors de CodeBlocks. J'ai donc installé les outils nécessaires pour utiliser un Terminal de Ubuntu sur Windows, appelé **Bash**.

Une fois le Dossier contenant les fichiers .cpp et .h atteint avec le terminale, il faut utiliser plusieurs commandes pour tout compiler.

La compilation des 3 fichiers (*.cpp , *.h et test_*.cpp) se fera en 4 étapes :

- Compiler le *.cpp, ce qui nous rendra un fichier *.o :

```
g++ -c *.cpp
```

-De même pour le test_*.cpp qui donnera un test_*.o

Des erreurs peuvent apparaître car le compiler utilisé par défaut par le terminale n'est pas une version assez récente, si une erreur de version apparaît, privilégier cette commande :

```
g++ -c -std=c++11 *.cpp
```

Ainsi on utilise la version C++11.

-On réunit ensuite nos deux fichiers résultats des compilations :

```
g++ *.o test_*.o -o NomDuPrgm
```

-On pourra enfin lancer le programme :

```
./NomDuPrgm
```

A chaque modification du programme, il sera nécessaire de répéter ces étapes pour en voir le résultat.

TESTS DES FONCTIONNALITES DE BASE

Test d'affichage

On va d'abord tester les fonctions d'affichage afin de les utiliser par la suite pour tester les autres fonctions. On ne va pas utiliser d'ASSERT pour les test, car ce sont des fonctions graphiques et il est plus naturel de lancer l'affichage et vérifier manuellement si il s'agit bien du résultat attendu.

Pour démarrer nos tests nous allons donc créer un fichier supplémentaire "test_affichage.cpp". Tous les fichiers de test commenceront par "test_....".

Dans notre fonction main, nous déclarons notre plateau. L'erreur vue précédemment qui empêchait le Compiler d'accepter la syntaxe suivante était dû à mon logiciel (Code::Blocks). Le compiler du logiciel n'était pas paramétré avec les dernières versions. Il fallait aller dans Settings > Compiler > cocher la ligne correspondant à [-std=c++11] ou [-std=c++0x] . Je vais donc pouvoir initialiser facilement tout un tas de tableau pour les tests via cette syntaxe :

```
plateau = { {1,2,3,4} , {2,3,4,5} , {3,4,5,6} , {4,5,6,7} };
```

```
*****  
* 64 * 256 * 512 * 2048*  
*****  
* 16 * 64 * 128 * 32 *  
*****  
* 2 * 4 * 16 * 4 *  
*****  
* 4 * 2 * 8 * 2 *  
*****
```

J'ai récupéré un plateau de jeu gagnant sur Internet que j'ai recopié dans mon test. Ainsi toutes les cases ont été testées, et toutes affichent correctement les nombres. Un seul détail à redire, lorsque le tableau n'est pas initialisé par la ligne ci-dessus, il faut l'initialiser autrement. Car sinon une erreur se produit lors de l'affichage du tableau "vide".

16 Nov :

Test des fonctions de Random Spawn

Je vais d'abord faire des grilles vides et lancer plusieurs fois les fonctions aléatoires afin de vérifier qu'elles fassent bien spawnner et des 2 et des 4 de façon aléatoire.

Je lance la fonction "eventProba1surN" avec une probabilité de $\frac{1}{2}$, j'en fais des séries de 10 tirages. Or chaque fois, il y a une chance sur deux pour que toute la série soit 0, ou toute 1. Il semblerait donc que le tirage au sort se fasse sur le premier tirage et que tous les autres soit le même. Il semblerait que cela ne se produise que lorsqu'on lance plusieurs fois de suite le programme avec une même probabilité.

La **solution** aurait pu être de lancer le programme avec une probabilité N, puis une autre probabilité, et enfin relancer avec N. Mais ceci ne marche pas car lorsqu'on demande 4 événements de probabilité 1, 1/4, 1 et 1/4, on obtient toujours soit 1010 soit 1111.

De plus, en lançant une série de tirage de probabilité 1/N avec N allant de 1 à 8, on obtient toujours une série de 1 (événement arrive) suivi d'une série de 0 (l'événement n'arrive pas). On en conclue donc que le programme n'actualise pas le nombre aléatoire à chaque fois qu'on lance la fonction mais à chaque fois qu'on lance main.

Tests à reprendre.

Test de Déplacements Partiels

Encore une fois, il est peu pratique d'utiliser des fonctions ASSERT pour ces tests, car le résultat visuel est bien plus satisfaisant et pratique à approcher. Cependant, afin pour garder une trace de tous les tests effectués, je vais m'y mettre.

Pour chaque fonction de déplacements partiels, je vais créer un tableau dont les valeurs seront placées de façon à ce que les 4 lignes correspondent à :

- 2222 : Une ligne pleine qui ne doit pas bouger
- 0002 : Une ligne dont 3 cases sont vides et la case pleine doit être transportée de l'autre côté
- 2022 : une ligne où la seconde case est vide, donc 2 cases doivent bouger
- 0202 : une ligne où la Première et Troisième cases sont vide, donc une case se décale de 1, et l'autre de 2

J'initialise donc des plateaux en dehors du ASSERT pour éviter les erreurs, ce qui me donne :

```
void test_deplacementsPartiel(){
    Matrice t; Matrice y;
    t = { {64,256,512,2048} , {2,0,0,0} , {2,16,0,4} , {4,0,8,0} };
    y = { {64,256,512,2048} , {0,0,0,2} , {0,2,16,4} , {0,0,4,8} };
    ASSERT( y == deplacementPartielDroite(t) );
}
```

Les tests ont permis de déceler une erreur, dans la fonction deplacementPartielBas, la dernière colonne était oubliée, il a fallu remplacer `j=0;j<3;j++` par `j=0;j<4;j++` .

Test de Fusion

Après les tests de déplacements partiels viennent ceux de la fusion. Ils ont permis de voir qu'après une fusion, le déplacement ne se faisait pas. Les nombres fusionnaient correctement, mais la case alors libérée n'était pas occupée par le numéro qui aurait venir.

Le problème venait de la condition if, elle ne faisait qu'appeler la fonction de déplacement, mais ne l'affectait pas au plateau.

Les tests se font selon les schémas suivant (0 aucun nombre, 1,2,3,4 quatre nombres) :

- 1224 --> 0134
- 0011 --> 0002
- 1122 --> 0034
- 1222 --> 0123

Screenshots de la fonction test :

```
-- Verifier que les affichages suivants sont correctes --

Affichage d'une separation :
*****

Affichage d'une ligne / 4 cases :
* 4 * 2 * 8 * 2 *

Affichage d'un tableau complet (gagnant) :
*****
* 64 * 256 * 512 * 2048*
*****
*      * 128 * 32 *
*****
* 2 *      * 16 * 4 *
*****
* 4 * 2 * 8 * 2 *
*****

Verifier qu'il n'y a pas de messages d'erreurs dans les 2 tests suivants :

-- TESTS DE DEPLACEMENTS --

--- TESTS DE SITUATIONS ---

-- Test sur les parametres aleatoires --
Verifier que les nbr suivants sont aleatoires et differents
a chaque nouveau tirage/lancement du test :
281238272
901922432
702842048
1156693376
1802640640

Verifier que dans la serie suivante,
1 et 0 apparaissent avec une proba de 1/2 :
1 0 1 1 0 1 0 1 1 0 1 1 1 1 1
```

```
Verifier que dans la serie suivante,
les 1 apparaissent avec une proba de 1/10 :
1 1 0 0 0 0 0 1 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Verifier qu'il n'y a pas d'erreurs a la ligne suivante :

Verifier que les 2 et les 4 apparaissent avec des probas
respectives de 9/10 et 1/10 :
2 2 2 4 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2

Verifier que le tableau suivant affiche 3 nombres aleatoirement :
*****
* 2 *      *      *
*****
*      * 2 *      *
*****
*      *      *      *
*****
*      *      *      *
*****

-- Affichage d'un plateau de jeu initial --

r : Restart
Ctrl+C : Exit

Mouvements :
z : Haut
q : Gauche
s : Bas
d : Droite

*****
* 2 *      *      * 2 *
*****
*      *      *      *
*****
*      *      *      *
*****
*      *      *      *
*****
*      *      *      *
*****
```

Changement dans la structure des dossiers

Afin de simplifier la structure des dossiers / fichiers et ne pas avoir à recompiler tous les fichiers dès qu'on veut refaire des tests (avec chaque fichier), je vais revoir la structure de mes fichiers. Je vais créer un nouveau dossier qui contient un nombre plus limité de fichiers.

EDIT : Par la suite, on utilisera un Makefile qui simplifiera la compilation.

17 et 18 Nov

Après avoir réarrangé tous les fichiers, je finis d'écrire tous les tests. Il y a parfois quelques petits détails à rectifier dans les fonctions mais rarement grand-chose.

Les fonctions "mouvmtPossible[...]" qui testent si un mouvement est possible dans une des directions vont être modifiées. La fonction qui test le haut et le bas **sont en fait les mêmes**, elles vont donc fusionner en une seule, de même pour Droite et Gauche. Car si un mouvement est possible dans une direction, il est forcément possible dans la direction opposée.

EDIT : Ce principe n'est en fait pas vrai. Si la seule case libre se situe sur la ligne du haut et qu'aucune fusion n'est possible, un mouvement vers le haut est possible mais pas vers le bas. Erreur rectifiée en page suivante.

Dans le fichier Test_..... .h, **toutes les fonctions sont listées sans documentations**. Dans le fichier StructureMinimale.h, **seules les fonctions principales** (celles qui seront mentionnées par le main du 2048.cpp) sont listées et documentées.

19 Nov

J'ai **terminé toutes les fonctions nécessaires au squelette principal** du jeu (Niveau 1 atteint). J'ai ensuite fait mon main dans mon fichier principal (2048main.cpp) un appel à une fonction jouerUnCoup qui se trouve dans ce même fichier. Cette fonction s'appelle elle-même, ainsi par **récurtivité**, on peut jouer des coups à l'infini.

LE JEU FONCTIONNE

Le programme est fonctionnel. Il peut ainsi :

- Insérer un 2 ou un 4 aléatoirement dans une case vide avec les probabilités liées de 1/10 et 9/10
- Lancer une partie avec 2 cases aléatoires de 2 ou de 4 respectant la règle précédente
- Lancer les 4 mouvements (droite, gauche, haut ou bas)
- Si le jeu est bloqué (=perdu) ou gagné, un message l'indique et une nouvelle grille apparaît
- Il est possible de quitter le jeu ou relancer une nouvelle grille avec Ctrl+C ou "r"

Après plusieurs parties, voici les **bugs à corriger** :

- Quand le jeu est bloqué vers un côté, et que ce côté est demandé, le jeu ne bouge pas mais fait apparaître un nombre.
- Quand le jeu est totalement bloqué, et qu'une commande est demandée, il ne répond plus.

Je pense qu'en fait rassembler les 4 fonctions test de movmntPossible en 2 n'était pas une bonne idée... A rectifier.

De plus le fichier .h contient toutes les fonctions de mon fichier annexe.cpp. Il faudra donc l'épurer, ne garder que les fonctions principales, et les documenter.

20 Nov

Je pense avoir compris **d'où venaient les bugs** précédents. Les fonctions movmntPossible renvoyaient True pour des raisons assez étranges (je n'arrivais pas à comprendre ce que j'ai voulu faire même après plusieurs relectures de ma fonction). Puis, j'ai finalement compris, un mouvement (par exemple vers le **haut**) n'est possible qu'à deux conditions :

- une case **non vide** possède une case vide au-dessus
- une case **non vide** possède une case de même valeur au-dessus

Or dans ma fonction, seule la première condition était intégrée, et la deuxième est mal interprétée. Je la corrige et relance les tests. Les **tests étaient donc également incorrects** puisqu'ils ne m'ont pas alerté du problème !

Je vais donc devoir re-séparer les tests movmntPossible en 4 tests, chacun une direction.

Ce qui pourrait également être modifié est l'affichage de "mouvement impossible" qui se fait au-dessus du nouveau plateau de jeu.

A présent, le jeu fonctionne, et est prêt à recevoir des extensions.

Screenshot du jeu du 20 nov

```

Entrer la commande :

q

r : Restart
Ctrl+C : Exit

Mouvements :
z : Haut
q : Gauche
s : Bas
d : Droite

*****
*      *      *      *      2      *
*****
*  4  *  2  *      *      *
*****
* 16  * 32  * 16  *      *
*****
* 64  * 128 * 64  * 32  *
*****

Entrer la commande :

```

LES EXTENSIONS - SCORE

Le **système de score** sur la version classique fonctionne ainsi :

- Lors de la fusion de deux cases, la valeur de la nouvelle case créée s'ajoute au score. Ainsi si le score est de 50 et que deux cases 8 se rencontrent, le score passe à 66 ($50 + 2 \times 8$).

C'est un système assez simple à rajouter, il faut que la fonction "grilleVide" qui initialise le tableau de jeu créer une cinquième ligne au tableau 2D, mais qui ne fait qu'une case de longueur. On l'initialise à 0. Ensuite, à chaque fusion, le score prend la valeur de lui-même ajouté à la valeur de la nouvelle case. Il a donc suffit ensuite de rajouter une ligne dans chaque fonction "fusion" :

```
plateau[4][0] += plateau[i][j];    // Augmente le score
```

Je vais changer le système d'affichage pour que, à chaque coup, s'affiche dans cet ordre : Le plateau de jeu, Le score, l'entrée de commande. Il faut également changer la fonction qui vérifie si le jeu est gagné, car lorsque le plateau est gagnant, il est directement réinitialisé à 0, alors **qu'il faudrait qu'il affiche 2048 ainsi que le score** avant de proposer une réinitialisation.

LES EXTENSIONS - NCURSES

Intro :

NCurses est sûrement ce qui m'a donné le plus de mal dans ce projet. Jusqu'à aujourd'hui (le 30 novembre), j'ai pris une dizaine de jour pour coder le 2048 (avec score) et il m'a fallu une dizaine de jours supplémentaires pour commencer à comprendre la librairie.

Une fois installée, j'ai eu beaucoup de mal à démarrer directement. Afin d'avancer pas à pas, j'ai regardé plusieurs vidéos de cours/découverte de plusieurs source/auteurs, j'ai également étudié les 2 documentations en lien dans le PDF de consigne, ainsi que plusieurs cours en ligne (en français et en anglais).

Fichier annexe

Je me suis petit à petit construit mon propre dossier nCurses afin d'avoir un fichier clair où je m'y retrouve mieux et où toutes les fonctions que j'utilise sont répertoriées et expliquées. Tout ceci se trouve dans un fichier annexe que j'appellerai "*Documentation Perso de NCurses pour le 2048 en C++.pdf*" une fois terminé.

Ce document présente plusieurs heures de découverte/étude de la librairie en autodidacte. Je ne détaillerai donc rien à propos de ncurses dans ce fichier.

Avec toutes les modifications régulières de chaque fonction, les tests ne sont plus valables. Ils seront donc repris à la fin.

Quelques commentaires sur la version actuelle

Le jeu est à présent terminé, nous sommes le 1^{er} décembre. Il ne contient plus de bug. J'y ai intégré le système de score, des couleurs, l'utilisation des flèches plutôt que d'entrer une commande, ainsi qu'un plateau de jeu fixe dans la console, qui se rafraichit à chaque coup. Il sera bientôt temps de remettre à jour le fichier de tests. Les documentations du .h ont été mises à jour.

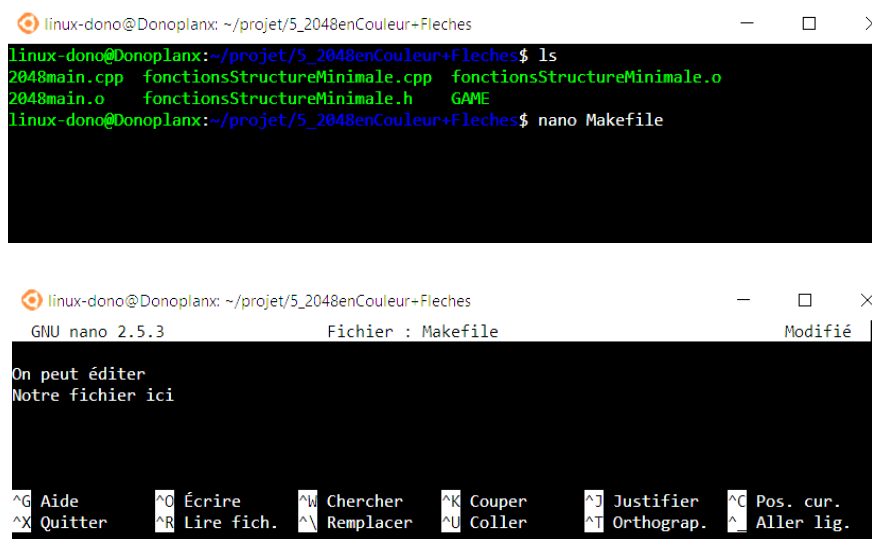
Mon travail ne respecte pas scrupuleusement les consignes du pdf. Entre autre, je n'ai pas les fonctions `const int`, `déplacement`, `string dessine`, `bool estTermine`, ni `int score`. De plus j'ai nommé mon `typedef vector<vector<int>> Matrice` et non `Plateau`, afin de bien différencier le type `Plateau` et la variable `plateau`.

MAKEFILE ET GIT

Avant de créer une variante du jeu dont j'ai déjà commencé à imaginer les règles (une variante qui à priori, n'existe pas encore), je souhaite **faciliter mon codage**. Les deux prochaines étapes tombent parfaitement bien. Nous allons premièrement nous pencher sur la création d'un Makefile, qui permet d'automatiser les compilations. Ensuite, nous installerons GIT, un logiciel gestionnaire de versions. Ainsi il me permettra d'avancer pas à pas et de me simplifier les sauvegardes et la gestion de tous mes dossiers.

CREATION DU MAKEFILE

Pour créer mon Makefile, je vais utiliser un outil de la console que j'ai déjà utilisé précédemment sans en parler, il s'agit de Nano. **Nano** est l'équivalent d'un éditeur de code (comme Dreamweaver ou NotePad) mais celui-ci est directement intégré à la console. Ainsi en tapant `nano nomdufichier.cpp` on peut modifier notre fichier en console. On va donc entrer la commande `nano Makefile`, ce qui va créer un fichier vide "Makefile".



```
linux-dono@Donoplanx: ~/projet/5_2048enCouleur+Fleches
linux-dono@Donoplanx:~/projet/5_2048enCouleur+Fleches$ ls
2048main.cpp  fonctionsStructureMinimale.cpp  fonctionsStructureMinimale.o
2048main.o    fonctionsStructureMinimale.h    GAME
linux-dono@Donoplanx:~/projet/5_2048enCouleur+Fleches$ nano Makefile

GNU nano 2.5.3      Fichier : Makefile      Modifié
On peut éditer
Notre fichier ici

^G Aide      ^O Écrire    ^W Chercher  ^K Couper    ^J Justifier  ^C Pos. cur.
^X Quitter   ^R Lire fich.^_ Remplacer  ^U Coller    ^T Orthograp.^_ Aller lig.
```

Ecriture du Makefile

Contrairement à d'habitude, pour le Makefile, je vais d'abord écrire mon fichier, et l'expliquer ensuite.

On s'intéressera ici à une forme très simple du Makefile, en raison de la simplicité de notre programme (seulement 2 fichier CPP et un .h). Je ne suis pas très satisfait de mes explications qui suivent, je m'excuse par avance si ce n'est pas très clair.

Screenshots en page suivante.

Mon Makefile

```
all: 2048main.o fonctionsStructureMinimale.o
    g++ 2048main.o fonctionsStructureMinimale.o -o GAME -lncurses

2048main.o: 2048main.cpp
    g++ -c -std=c++11 2048main.cpp -lncurses

fonctionsStructureMinimale.o: fonctionsStructureMinimale.cpp
    g++ -c -std=c++11 fonctionsStructureMinimale.cpp -lncurses

clear:
    rm *.o
```

Structure du Makefile

On remarque une structure dans le fichier, construite ainsi :

```
cible: dependance
      commandes
```

Cible est ... La cible de notre commande. Elle peut être la création de notre fichier exécutable (all), un nom de fichier compilé (main2048.o), ou bien la commande clear (qui supprime les fichiers intermédiaires dans mon cas).

La commande suit la cible et la dépendance, elle est précédée d'une tabulation, sans cette dernière, le fichier ne reconnaît pas que c'est une commande.

La dépendance va déterminer si la commande sera exécutée. La dépendance cible un fichier, si celui-ci a été modifié depuis la dernière fois, alors les instructions seront exécutées.

Fonctionnement du Makefile

Le fichier va d'abord regarder notre première dépendance, si un des fichiers 2048main.o ou fonctionsStructureMinimale.o a été modifié depuis la dernière fois, alors l'instruction suivante (la compilation finale) sera lancée. Or la dépendance va aller chercher plus loin dans les cibles. Notamment, la cible 2048.o dépend de 2048main.cpp. Cela veut dire que, de même que précédemment, si 2048main.cpp a été modifié récemment, alors l'instruction est lancée.

En résumé, imaginons que le fichier main.cpp a été modifié, alors l'instruction liée à la cible main.cpp sera lancée et formera un fichier main.o. Le fichier main.o va donc se mettre à jour, et ainsi, la cible all aura l'ordre de se lancer.

Clear ne contient aucune dépendance, et est situé à la fin. Ainsi pour être lancé, on doit rentrer la commande `make clear`.

`rm *.o` peut se traduire par ReMove (supprimer) *.o (tout ce qui finit par .o).

Version finale du Makefile

```
all: GAME_LAUNCHER

GAME_LAUNCHER: 2048main.o fonctionsStructureMinimale.o
    g++ 2048main.o fonctionsStructureMinimale.o -o GAME_LAUNCHER -lncurses
    rm *.o

2048main.o: 2048main.cpp fonctionsStructureMinimale.h
    g++ -c -std=c++11 2048main.cpp -lncurses

fonctionsStructureMinimale.o: fonctionsStructureMinimale.cpp fonctionsStructureMinimale.h
    g++ -c -std=c++11 fonctionsStructureMinimale.cpp -lncurses
```

Ce qui se passe au lancement de mon Makefile

Lorsque je rentre ma commande `make`, voici ce qui se passe dans mon cas :

-Il voit `all` qui dépend de `GAME_LAUNCHER`, il va donc à la cible `GAME_LAUNCHER` (qui est le bloc suivant). Celui-ci contient une commande `rm *.o`, qui ne crée aucun fichier de sortie (ce n'est pas une commande de compilation). Par conséquent, il sera toujours lancé.

Un bloc d'instruction qui ne contient que des compilations n'est lancé que si les fichiers dont il dépend ont été modifiés depuis la dernière compilation.

-`GAME_LAUNCHER` dépend des deux fichiers `*.o`, donc ces cibles sont trouvées dans les 2 blocs suivants et sont exécutées.

-Ordre : Le makefile va d'abord chercher les cibles les plus lointaines dans les chemins de ciblage/dépendance. Il va donc d'abord compiler les `*.cpp` et en faire des `*.o`.

Ensuite il entre les commandes de `GAME_LAUNCHER` qui compilent les `*.o` en un fichier exécutable, et qui supprime les `*.o` qui sont des fichiers temporaires.

Les fichiers `.o` n'ont aucune utilité autre que la création du fichier exécutable. Une fois ce dernier créé, ils peuvent être supprimés. Ce sont donc des fichiers temporaires.*

```
linux-dono@Donoplanx:~/projet/5_2048enCouleur+Fleches$ ls
2048main.cpp  fonctionsStructureMinimale.cpp  fonctionsStructureMinimale.h  Makefile
linux-dono@Donoplanx:~/projet/5_2048enCouleur+Fleches$ make
g++ -c -std=c++11 2048main.cpp -lncurses
g++ -c -std=c++11 fonctionsStructureMinimale.cpp -lncurses
g++ 2048main.o fonctionsStructureMinimale.o -o GAME_LAUNCHER -lncurses
rm *.o
linux-dono@Donoplanx:~/projet/5_2048enCouleur+Fleches$ ls
2048main.cpp  fonctionsStructureMinimale.cpp  fonctionsStructureMinimale.h  GAME_LAUNCHER  Makefile
linux-dono@Donoplanx:~/projet/5_2048enCouleur+Fleches$
```

INSTALLATION DE GIT

On va maintenant installer GIT, le logiciel gestionnaire de versions. Ainsi je vais pouvoir **sauvegarder toutes mes versions plutôt que de recréer un nouveau dossier à chaque fois**. Je vais pouvoir plus aisément apporter des améliorations à mon code, sans changer le fonctionnement, avant de passer à la suite. En bref, je souhaite optimiser mon code et/ou le rendre plus clair.

Je vais donc, après avoir installé GIT, et avant de passer à la suite (création d'une variante), installer **une touche permettant de revenir un coup en arrière**. Je vais aussi créer une fonction qui permet de stocker le plus haut score (**Highscore**) dans un fichier externe.



Qu'est-ce qu'un gestionnaire de versions ?

Afin de pouvoir travailler aisément à plusieurs sur un même projet informatique, ou simplement pour se simplifier le codage, on peut utiliser un gestionnaire de versions. Celui-ci va sauvegarder chaque modification apportée à un projet au fur et à mesure que celui-ci est travaillé.

Ainsi, si l'on se rend compte que notre programme entraîne une erreur qui n'était pas là avant, on peut revenir à la version correcte grâce au gestionnaire.

De plus, lors d'un travail en groupe, le logiciel permet de travailler à plusieurs sur un même fichier. Ainsi, chacun peut voir quelle partie a été codée par qui, et dans quel but. De plus, il permet que la modification de l'un n'écrase pas les modifications de l'autre si les 2 développeurs travaillent simultanément.

J'ai donc compris qu'il y avait 2 types de gestionnaire de versions.

Les **centralisés**, où toutes les modifications sont sauvegardées par un serveur externe. Pour modifier le fichier, chaque développeur va alors se connecter au serveur.

Sinon, il y a les **distribués**, dans ce cas, il n'y a pas de serveur externe. Chaque développeur possède la dernière version des fichiers, et possède toutes les sauvegardes sur sa machine. Pour travailler à plusieurs, ils sont donc tous reliés entre eux, à la façon du peer-to-peer.

On peut distinguer deux formes d'enregistrements. Une qui enregistre **le projet entier**, qui prend de la place, et qui copie une grande partie du projet qui n'a pas subi de modifications, elle est donc assez lourde. Une qui n'enregistre **que les modifications**, plus légère, mais qui ne rend pas compte du projet dans son ensemble.

COMMANDES DE GIT

Voici les premières commandes qui nous serviront notamment à lancer nos premières sauvegardes.

Voici le déroulement, on va créer notre GIT local, y mettre des fichiers, et enregistrer les modifications (créer un "commit", une sauvegarde).

Mais premièrement, il faut configurer notre GIT avec notre nom et adresse mail :

```
git config --global user.name "Donovan"  
git config --global user.email donovant78@gmail.com
```

Créer notre dossier GIT :

```
git init 2048ProjetGIT
```

Créer un dossier "2048ProjetGIT" qui contient toutes les infos et commits GIT.

```
cd 2048ProjetGIT
```

On va dans ce dossier.

A présent, tous les fichiers que l'on souhaite gérer avec GIT devront être mis dans ce dossier. Imaginons que j'y place mon fichier file.cpp.

```
git add file.cpp
```

Ceci indique à GIT que file.cpp devra être inséré dans la prochaine sauvegarde.

Pour sauvegarder notre projet, on va alors entrer la commande :

```
git commit -m"Message ou Commentaire ici"
```

Cette commande sera à lancer chaque fois que l'on veut entraîner une sauvegarde.

On peut visionner notre historique de commits avec la commande :

```
git log
```

Les fichiers modifiés depuis le dernier commit sont listés par :

```
git status
```

Chaque modifications depuis le dernier commit est détaillé par :

```
git diff
```

A chaque commit, on doit à nouveau Add les fichiers du commits. Si ceux-ci sont les mêmes que le commit précédent, on peut utiliser la commande :

```
git commit -a
```

Pour un affichage du fichier avec en rouge tout ce qui a été supprimé et vert rajouté, on utilise `git log -p` et pour naviguer dans les fichiers, on utilise les flèches (haut et bas) et Q pour quitter.

Supprimer un commit et modifier les fichiers (revenir à la version d'avant) :

```
git reset --hard HEAD^
```

Faire revenir un fichier à sa version précédente :

```
git checkout nomdufichier
```

COMMANDES DE GIT (SUITE)

Pour avoir un aperçu des changements, on peut utiliser

```
git log --stat
```

```
linux-dono@Donoplanx:~/projet/6_GIT2048$ git log --stat
commit 02cdf133c3677200997298e6a1adc0371b89f007
Author: Donovan <donovant78@gmail.com>
Date:   Sun Dec 3 23:51:37 2017 +0100

    Conversion en struct jeuGlobal, erreur de std::bad_alloc et core dumped

2048main.cpp | 34 ++--
fonctionsStructureMinimale.cpp | 313 ++++++-----
fonctionsStructureMinimale.h | 65 +++++
3 files changed, 208 insertions(+), 204 deletions(-)

commit 6b79a903a4deb529fe03d7ec88544e0ccd40be1e
Author: Donovan <donovant78@gmail.com>
Date:   Sun Dec 3 18:31:37 2017 +0100

    Toute première Version Originale Avec NCurses

2048main.cpp | 83 +++++
Makefile | 15 +
fonctionsStructureMinimale.cpp | 685 ++++++
fonctionsStructureMinimale.h | 135 +++++
4 files changed, 918 insertions(+)
```

Supprimer le dernier commit (sans changer les fichiers) :

```
git reset HEAD^
```

Restaurer la version d'un fichier datant du commit précédent :

```
git checkout nomdufichier
```

Les branches

Afficher les branches :

```
git branch
```

Créer une branche :

```
git branch nomdelabranche
```

Basculer vers une autre branche

```
git checkout nomdelabranche
```

Fusionner les changements d'une branche avec la master (en se plaçant sur la master) :

```
git merge nomdelabranche
```

Supprimer une branche dont les changements ont été fusionnés :

```
git branch -d nomdelabranche
```

Supprimer une branche dont les changements ne sont pas fusionnés :

```
git branch -D
```

LES TYPES STRUCT

Afin d'**améliorer notre code et le préparer à la suite**, nous allons utiliser les Structures.

La structure (utilisée pour créer des variables structurées) va nous permettre d'utiliser un élément, un "type" (au même titre que String, Char, Double...) mais qui va en contenir plusieurs. Je m'explique un peu plus bas.

Syntaxe pour déclarer notre variable structurée

```
struct nomStructure { //initialisation
    type1 champ1;
    type2 champ2;
    ...
};
struct nomStructure nomVariable; //déclaration
```

Pour appeler un champ, on utilise ensuite `nomVariable.champ`.

Dans notre cas, nous allons créer une structure que l'on va appeler `jeuGlobal`. Celui contiendra notre plateau (de type Matrice) et le score (de type int). Nous allons donc séparer le plateau de jeu et le score. Voici donc le code :

```
struct jeuGlobal {
    Matrice plateau;
    int score;
};
struct jeuGlobal jeu;
```

On peut alors initialiser notre jeu :

```
jeu.plateau = { {0,0 ... 0,0} };
jeu.score = 0;
```

On pourra ensuite changer notre augmentation du score en changeant ...

```
plateau[4][0] += plateau[i][j] ;
... par :
jeu.score += plateau[i][j];
```

Il faudra cependant faire attention, certaines fonctions ne renvoient plus un type Matrice mais un `jeuGlobal`. Ce sont celles qui sont susceptibles de modifier à la fois le score et le plateau. Si elles ne modifient que le plateau, elles pourront rester de type Matrice.

Durant mon codage, je vais "Rechercher et remplacer tout" de "`Matrice plateau`" à "`jeuGlobal jeu`" afin d'automatiser pas mal de remplacements. Dreamweaver permet également d'écrire sur plusieurs lignes en même temps. C'est pratique lorsque l'on doit remplacer ou écrire plusieurs fois le même nom de variable.

Je remarque aussi que mes fonctions `deplacementsCoté` possède un `if(mouvmtPossible)` inutile, puisque ce test est déjà fait auparavant. Je peux donc supprimer ces tests.

CHANGEMENTS MAJEURS – AJOUT DES STRUCTS

On opère des modifications dans quasiment chaque fonction puisque toutes les variables "plateau" doivent être convertis en "jeu" (jeu.plateau pour être précis).

Après toutes les modifications, comme on pouvait s'en douter, **on obtient une longue liste d'erreurs**. Je me demande s'il ne faut pas préciser "struct" devant chaque "jeuGlobal". On va essayer.

Je remarque qu'une grande partie des erreurs (une trentaine environ) proviennent du fichier .h. Les messages sont soit " 'jeuGlobal' does not name a type" ou " 'jeuGlobal' was not declared in this scope".

Solutions :

Beaucoup des erreurs étaient dues au fait que "struct jeuGlobal" n'apparaissant que dans l'en-tête du 2048main.cpp et pas dans les autres fichiers. Chaque fois que jeuGlobal était écrit ailleurs, une erreur apparaissait. On peut alors résoudre cela en l'écrivant dans le fichier.h, on le supprime ensuite du 2048main.cpp, sinon il y a une redéfinition.

Les erreurs suivantes, liées au fichier 2048main.cpp sont simplement des fautes de frappes ou petites erreurs d'inattention (des "plateau" qui n'ont pas été changés en "jeu" par oubli).

De même pour le fichier (annexe).cpp, la plupart des erreurs étaient dues à la redéfinition de Struct jeuGlobal ou à des erreurs d'inattention.

Nouvelle erreur :

Quand je lance mon fichier exécutable, voici ce que j'obtiens :

```
terminate called after throwing an instance of 'std::bad_alloc'      what():  std::bad_alloc
Abandon (core dumped)
```

Il semblerait que mon programme nécessite trop de mémoire... Peut-être est-ce dû à ma fonction récursive "jouerUnCoup" qui s'appelle en boucle.

Afin de résoudre ce problème, je vais modifier la structure de mon code. Je vais d'abord rectifier la fonction aléatoire qui n'est pas optimisé. Je vais ensuite revoir la structure de mon main.cpp. Celui utilise une fonction récursive à l'infini (elle s'appelle elle-même sans jamais se terminer).

Au lieu de cela, je vais utiliser une fonction qui renvoie le nouveau tableau, qui affiche toutes les infos nécessaires, et qui renvoie le nouveau tableau. Ainsi, je pourrai mettre une boucle while dans mon main.cpp qui tourne à l'infini, et alors je n'aurai plus de fonction récursive.

EDIT : Après résolution du problème, on comprend que l'affichage bug (texte qui commence n'importe où). C'est dû au fait que l'erreur arrive avant la fin du programme, ce dernier s'arrête donc avant de pouvoir se terminer. Donc la fenêtre de ncurses se ferme pas (avec le endwin).

PLUSIEURS AMELIORATIONS VITALES

J'en suis à présent rendu à un point où je ne peux continuer sans épurer/améliorer certaines parties de mon code. Je vais donc me pencher là-dessus avant de continuer et rajouter mes règles de variante.

L'aléatoire

(Changé sous les conseils du professeur chargé de TD)

Voici le fonctionnement actuel de ma fonction randomspawn (qui fait apparaître un nombre aléatoirement sur le tableau). Elle scan le tableau à la recherche de cases vides. Si la case n'est pas vide, on passe à la suivante. Dès qu'elle en trouve une vide, on lance un test de probabilité pour savoir si la case va accueillir le prochain nombre. La probabilité est de $1/(\text{nbrDeCasesVides})$. Dès qu'une case reçoit un nombre, le plateau est *return* et donc la fonction s'arrête.

Le problème ?

Ce système fait que chaque case vide du tableau va être scannée, et testée avec la probabilité. Si la probabilité est petite (qu'il y a beaucoup de cases vides) alors le scan peut durer longtemps (plusieurs fois le tableau), ce n'est pas optimal.

On va donc choisir aléatoirement un nombre aléatoire **N** entre 1 et le nombre de cases vides. La fonction va ensuite scanner chaque case vide **jusqu'à tomber sur la N^{ième} case vide**, ce sera celle qui accueillera notre nombre.

Nous utiliserons GIT pour l'écriture de nos améliorations. On veille donc à bien gérer les commits et les branches avant de commencer les modifications du code.

J'avais jusqu'à maintenant enregistré un commit pour ma version d'origine sur GIT, et ensuite un commit pour la version qui bug sur la branche principale (master). En bref, je n'ai pas utilisé GIT correctement. Je vais donc supprimer ce deuxième commit, créer deux branches annexes **amelio_aleat** et **amelio_structJeuGobal** dans lesquels je vais faire les modifications, pour les fusionner à la branch master une fois fonctionnel.

Même après correction de la partie aléatoire, le programme ne démarre pas correctement. Il ouvre la fenêtre de ncurses mais vide, et impossible d'y écrire quoi que ce soit, elle reste vide. Je supprime la ligne 542 de mon fichier annexe.cpp qui se situe dans la fonction randomspawn (une boucle while inutile). Je recompile, lance le tout, et **j'obtiens à nouveau mon fameux message "Erreur de segmentation (core dumped)"**.

Je vais donc devoir revoir la structure de mon code pour contourner cette erreur, avant de pouvoir passer à la suite.

Je vais donc créer une troisième branche sur GIT (autre que la master) et commencer à tout revoir.

Je revois toute la structure

J'ai essayé de revoir ma structure dont j'ai parlé en fin de page 32.

Une fois ceci fait, j'ai compilé et lancé le programme... Toujours le même problème d'Erreur de segmentation.

A LA RECHERCHE DE LA SOURCE DE L'ERREUR

Forcé à dénicher l'erreur pour pouvoir continuer. Je cache une à une chaque partie de mon programme, dans la fonction `main` au début, afin de voir d'où provient l'erreur. Pour cela, j'utilise les commentaires de plusieurs lignes avec `/** */`. C'est cela que j'appellerai parfois "code désactivé" ou "inactif", une partie de code mit sous commentaire.

Premièrement, je vois que dans le `main()`, **l'erreur apparaît directement** lors de l'exécution de la fonction `"plateau = plateauInitial(plateau)"`.

Mais ce n'est pas tout, et l'erreur qui vient est bien étrange. Une fois la fonction `plateauInitial` caché, "Erreur de segmentation (core dumped)" s'affiche **uniquement si les 2 lignes 47 et 48 sont actives**.

Autrement dit, quand `plateau = commandeExecuter(commande, plateau);` et `plateau = testsDeJeu(plateau);` sont exécutées à la suite.

Si on désactive une des deux lignes (en la transformant en commentaire) alors aucune erreur n'apparaît.

A l'aide de `printw` et de `getch`, je m'aperçois que l'erreur ne se fait pas lors de `commandeExecuter`, mais une fois que ceci est passé, lors de l'exécution de `testsDeJeu`.

On peut donc penser que cette erreur peut être due à ce que `commandeExecuter` renvoie, que `testsDeJeu` n'arrive pas à supporter.

Etant donné que le plateau fait $4 \times 4 + 1$ case, je me demande s'il n'y a pas une fonction qui n'apprécie pas cela.

Si on désactive tout le contenu de `commandeExecuter`, alors aucune erreur ne se produit. Si on active seulement le début (Restart et Haut), et que l'on demande un restart ou un Haut à chaque coup, alors `testsDeJeu` entraîne l'erreur. On se rapproche de la source, on sait maintenant que c'est dans cette partie qu'elle se cache.

Là encore, l'erreur n'apparaît **que si** les deux lignes `"plateau=delacementHaut(plateau)"` et `"plateau=randomspawn(plateau)"` sont actives. Si l'on désactive une des deux lignes, aucune erreur n'apparaît.

Or **lorsque seul randomspawn est active**, on obtient un score totalement incompréhensible, ainsi qu'une grille contenant de fausses valeurs (pas des puissances de 2).

(Screenshot en page suivante)

```

+-----+
|      |      |Erreur :Valeur de la case incorrect|
+-----+
|Erreur :Valeur de la case incorrect|      |1024|
+-----+
|Erreur :Valeur de la case incorrect|      |      |
+-----+
|Erreur :Valeur de la case incorrect|      |      |
+-----+

```

SCORE : 23284640

Ctrl+C : Quitter
R : Restart

Utiliser les fleches pour se deplacer

Je change alors la fonction d'affichage pour qu'elle m'affiche les vraies valeurs de cases. Une fois rendu là je demande plusieurs coups vers le haut, je retrouve finalement ma fameuse erreur :

```
+-----+
|16070324| |449| |
+-----+
|16084032| |16079456| |
+-----+
|16084032| |1851673464|32642|
+-----+
|16084032| |1851673464|32642|
+-----+
```

SCORE : 16079604

Ctrl+C : Quitter
R : Restart

Utiliser les fleches pour se deplacer

CommandeExecuter faitterminate called after throwing an instance of 'std::bad_alloc'

```
what(): std::bad_alloc
Abandon (core dumped)
```

J'ai désactivé tout le contenu de ma fonction `randomspawn`, et en lançant mon programme, je m'aperçois que j'ai également désactivé mon `return`. Je m'aperçois donc, **par hasard**, que **l'absence de `return` entraîne un comportement semblable** à ce que faisait déjà ma fonction précédemment. Je peux donc penser que **la fonction `randomspawn` n'arrivait pas jusqu'au `return`**, ce qui entraînait un tableau qui contenait n'importe quoi.

Cela expliquerait pourquoi "plateauInitial" faisait immédiatement crash le programme, cette fonction fait appel à 2 fois randomspawn.

J'ai fait beaucoup de modifications jusqu'à maintenant afin de trouver mon erreur, j'enregistre donc tous les changements dans la branche de Git où je me trouve. Cette branche n'était peut être pas prévue pour cela initialement, mais maintenant elle servira de branche de débogage. Sachant cela, je réactive mon `plateauInitial`. Je continue de désactiver ma fonction par partie pour voir d'où vient le problème.

Il semblerait que le problème vienne de l'intérieur du `if` final.

Edit : Après avoir réactivé la fonction entière, tout fonctionne toujours correctement. Je ne comprends pas bien comment le bug a pu disparaître. De plus, j'ai remarqué que lorsque l'on réactive la boucle infinie et que l'on appuie en boucle sur R pour afficher un plateau initial à chaque coup, aucun numéro ne s'affiche excepté parfois la première case.

RESOLUTION DE L'ERREUR ♥

L'erreur provenait de plusieurs endroits à la fois... Cela m'a pris plusieurs heures afin de réussir à tout résoudre. La première partie du bug se situait dans la fonction `randomspawn` qui n'avait pas été correctement refaite lors de l'implantation du nouveau système de spawn (cf page 33).

De plus, la fonction `numCasePrSpawn` qui était censée déterminer un entier entre 1 et le nombre total de cases vides. Celui-ci devait parfois diviser par 0. En effet il contient un $n-1$, avec $n \leq 1$, donc si $n=1$, alors on doit diviser par 0, et cela devait entraîner une erreur. De plus, l'entier qui sortait était parfois 0 ou même -1, ce qui désignait l'emplacement du nombre à spawner. La case -1 n'existe pas, donc rien n'apparaissait. Ce qui explique que parfois, un spawn n'apparaissait pas.

Dure dure la résolution

Bien que l'apprentissage de Ncurses et GIT furent très difficile durant ce projet, je pense que la **résolution de ce problème fut le plus gros défi**. J'ai bien faillit abandonner face à l'inconnu de la situation, et la difficulté à trouver l'origine du problème. Cette fois, ce n'était pas un sujet qu'il est possible d'apprendre, où il existe des cours sur Internet. Lorsque l'on recherche le nom de l'erreur que j'ai obtenu, on tombe souvent sur des programmeurs novices comme moi qui sont assez désespérés face à la situation. Si je suis fier de quelque chose dans ce projet, ce n'est pas d'avoir réussi à créer mon 2048 (et peut-être ma variante si j'ai le temps avant la date limite), mais d'avoir réussi à résoudre ce problème qui me paraissait insurmontable aux premiers abords.

Méthodes de résolution

Les premières étapes sont tester CHAQUE ligne afin de voir d'où peut provenir l'erreur. Bien que j'ai rapidement vu que la fonction `plateauInitial` posait problème, j'ai préféré passer outre cette fonction et voir la suite (car ce n'était pas la seule source du bug, même une fois désactivée, on obtenait l'erreur).

Et quelle étrange et mauvaise surprise j'ai rencontré en trouvant les 2 autres lignes de codes qui entraînaient l'erreur. Lorsque je testais la première ligne seule, la deuxième n'entraînait aucun bug. C'est donc la première qui posait problème ? J'active la première, désactive la seconde, et tout fonctionne encore sans erreur. **Le problème se situait dans le plateau erroné que la première ligne retournait, qui était utilisé par la seconde fonction, et cette dernière repérait qu'il y avait quelque chose d'anormal.**

L'erreur se situait avant tout dans la fonction `randomspawn` (même si ce n'est pas la seule source), voilà pourquoi la fonction `plateauInitial` entraînait aussi l'erreur. Cette fonction réinitialise un plateau vide, et lance 2 fois la fonction `randomspawn`.

Même après avoir compris que le problème venait du passage de la première fonction à la deuxième (et donc que le problème était situé quelque part dans la première fonction), ce n'était pas encore joué. Car j'ai suivi tout le chemin que faisait mon plateau dans cette fonction, et j'ai rencontré le même problème plus loin : 2 fonctions qui entraînent l'erreur que si les 2 sont actives. Il y a de quoi désespérer en tombant sur ça.

La deuxième étape de ma résolution, c'est de **sans cesse utiliser des printf et getch** afin de voir à quel moment arrive l'erreur, ou pour suivre l'évolution d'une variable. Ainsi j'ai pu voir parfois si l'erreur se faisait lors du return d'un plateau ou juste avant. J'ai également affiché "il y a **x** cases vides et la case **x** va accueillir le prochain spawn". C'est cela qui m'a permis de voir que parfois, la fonction qui return le numéro de case d'accueil, renvoi parfois un 0 ou un -1.

Ainsi ce système permet, au sein d'une fonction, de **voir à quelle étape le bug se produit**. Ou alors, il permet de **suivre l'évolution de la valeur de variables**.

Exemple :



```
// plateau de jeu
Matrice plateau;
plateau = { {2,8,32,4} , {16,64,2,8} , {8,16,32,4} , {2,8,4,2} , {99} } ;
if( movmtPossibleBas(plateau) and movmtPossibleDroite(plateau) and movmtPossibleGauche(plateau) ) {
    mvprintw(21,12,"Un mouvement est possibles");
} else {
    mvprintw(21,12,"Aucun mouvement n'est possible" );
    if( estBloque(plateau) ) {
        affichageJeu(plateau) ;
        mvprintw(22,12,"Test réussi : Le jeu est bloqué");
        refresh; getch();
    } else {
        affichageJeu(plateau) ;
        getch();
        mvprintw(22,12,"Test rate : Le jeu ne détecte pas le blocage");
        refresh; getch();
    }
}
```

Reprise des activités d'améliorations

Afin de continuer mon programme en vue de créer ma variante, j'ai maintenant trouvé la source de mes erreurs. Je vais donc faire un commit de cette version, puis tout épurer (retirer le code inutile comme les getch ou printf, et remettre en clair tout ce qui aurait pu être caché). Je vais ensuite fusionner cette fonction débuggée avec la branche master. De là, je pourrai créer d'autres branches pour passer à la suite.

Version épurée

J'ai épuré mon code, de tous les commentaires qui faisaient les tests, et de tous les `printw`. Il reste un dernier problème à corriger, la fonction "jouerUnCoup" prenait en compte un commentaire qu'elle affichait dans la version précédente (pour Mouvement impossible ou Mauvaise touche). Maintenant que les fonctions ont été modifiées, l'affichage de ces commentaires se font dans des fonctions annexes, et ne collent plus avec le clear, et sont effacés avant de pouvoir être affichées.

Affichage des commentaires

J'ai réparé le système de commentaire qui n'apparaissait plus. Pour cela, j'ai supprimé quelques lignes qui lançait la fonction `jouerUnCoup` à des moments non voulus, et j'ai changé l'ordre des fonctions dans `jouerUnCoup`.

Jeu terminé

Comme le montre les Screenshots (page précédente) j'ai quelques problèmes avec ma fonction `estBloque`. Celle-ci ne renvoie pas les infos correctement. Une fois corrigé, je supprime à nouveau toutes les lignes qui m'ont servi de tests.

Je vérifie ensuite que les fonctions principales sont à jour dans mon fichier `.h`, et je fais un dernier commit sur GIT. Je fusionne ensuite ce commit dans la master (on "merge" la branche).

Après la merge je m'aperçois que j'ai oublié de supprimer un des tests, je vois également qu'un getch doit être en trop quelque part après un restart, car il faut appuyer sur une touche Après R pour lancer un restart.

Avant de remettre en place mes types `Struct`, je souhaite mettre en place un fichier qui permet d'afficher le plus Haut score enregistré. Pour cela, il va créer un fichier annexe "highscore" qui contient uniquement un entier, qui sera la valeur du highscore. A chaque coup, le highscore devra (ou pas) se mettre à jour.

Il faudra également gérer l'affichage qui va se retrouver décaler d'une ligne.

Je garde en mémoire ma branche `changementStructureSansRecursivite` qui est celle sur laquelle j'ai travaillé toutes mes résolutions de bug, et je supprime les 2 autres qui n'ont pas abouti.

MA VARIANTE - HIGHSCORE

Je créer d'abord mon fichier vide "Highscore". Je pense à le Add dans GIT.

Voici l'idée du fonctionnement :

Dans les fonctions de fusion, on va ouvrir le fichier Highscore, lire la valeur, fermer le fichier. Si le highscore est plus petit que le nouveau score, on réécrit ce nouveau score dans Highscore. Sinon, il ne se passe rien de spécial.

De plus, dans l'affichage, juste après l'affichage du Score, on affiche le Highscore. On décale alors les commandes d'une ligne plus bas.



Et voici ce que l'on obtient après avoir relancé une partie :



Une fois finie, on enregistre un commit dans GIT, et on merge la branche vers la branche master. (J'ai fait une partie à 14 752 de score)

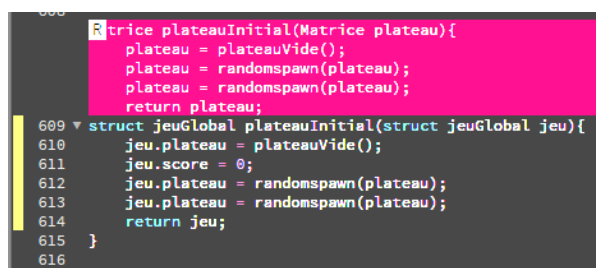
MA VARIANTE – STRUCT JEUGLOBAL LE RETOUR

Maintenant que tout mon code est fixe (nettoyé de toutes les possibles erreurs), je vais pouvoir me pencher à nouveau sur cette idée de strut, évoqué dans les pages 31 et 32.

Comme d'habitude, je créer une branche en dehors de la master pour effectuer toutes mes modifications.

Il y a beaucoup de modification à faire à la main, **il y aura sûrement quelques oublis**. On s'attend à avoir des erreurs.

Remarque : GIT associé à Dreamweaver permet de voir les changements. Ici, un bout de code changé. Je n'en ai pas parlé plus tôt mais c'est très pratique.



```
609 Matrice plateauInitial(Matrice plateau){  
610     plateau = plateauVide();  
611     plateau = randomspawn(plateau);  
612     plateau = randomspawn(plateau);  
613     return plateau;  
614 }  
615 struct jeuGlobal plateauInitial(struct jeuGlobal jeu){  
616     jeu.plateau = plateauVide();  
617     jeu.score = 0;  
618     jeu.plateau = randomspawn(plateau);  
619     jeu.plateau = randomspawn(plateau);  
620     return jeu;  
621 }
```

Dreamweaver affiche à gauche des numéros, par rapport au dernier commit :

- une bande **rouge** lorsqu'une ligne a été **supprimé**
- une bande **orange** lorsque la ligne a été **modifiée**
- une bande **verte** lorsque la ligne a été **insérée**

*Note : Je ne remplace pas absolument tout le programme. Par exemple, j'ai laissé le type Matrice pour la fonction déplacementPartiel. Alors la fonction fusion a été transcrite en struct jeuGlobal. Le choix du changement ou non se **décide en fonction de l'utilisation ou non du score par la fonction**. Dans déplacementPartiel, seul le plateau est susceptible de changer. Alors que dans la fusion, le score aussi change.*

Les bool mouvmtPossible, afficheLigne, estBloque, estGagne et randomspawn aussi restent inchangées.

On supprime la fonction grillePleine qui ne sert à rien puisqu'on a un équivalent nbrCasesVides.

C'est un exercice difficile de changer toute la structure, car **aucun test n'est possible tant que tous les changements n'ont pas été effectués**. Sinon, on risque de tomber directement sur des erreurs.

Une fois tous les changements effectués, on met à jour le fichier .h et on croise les doigts pour ne pas avoir oublié de changements.

Comme on s'y attendait, il y avait plein d'oublis (j'aurai presque été déçu de tout réussir du premier coup).

L'extension est maintenant rajoutée, on va pouvoir passer à la suite.

BUG : DES NOMBRES QUI DOUBLENT SANS RAISON

Si on part du tableau ci-dessous, en alternant Gauche, Droite, Gauche, Droite... On s'aperçoit qu'après quelques tours, le 32 de droite fini toujours par se transformer en 64 sans raison, sans fusion. Ce qui permet de débloquent le jeu et continuer... Je ne comprends pas bien cette action, je vais devoir mener une nouvelle enquête.

		4	
		8	
2	32	64	32
1024	512	128	64

Je fais de même en commençant par un mouvement vers le haut. On s'aperçoit que la fusion aléatoire apparaît toujours sur les mêmes cases lorsque l'on part de la même grille avec les mêmes mouvements (donc ce n'est pas aléatoire). De plus, cela arrive toujours lors d'un mouvement vers la gauche.

Ce n'est pas sûr mais il semblerait que cela n'ait lieu que sur la dernière colonne.

```
struct jeuGlobal fusionGauche(struct jeuGlobal jeu) {  
    int launchscan = 0;  
    for (int i=0 ; i<4 ; i++) {  
        for (int j=0 ; j<3 ; j++) {  
            if(jeu.plateau[i][j] == jeu.plateau[i][j+1]) {  
                jeu.plateau[i][j] *= 2;  
            }  
        }  
    }  
}
```

Résolution :

En effet, dans la double boucle for de la fonction déplacement, on trouvait $i < 4$ et $j < 4$. Or ce n'est possible car dans les fonctions de déplacement, il y a forcément une ligne ou une colonne qui n'est pas scannée. On remplace donc le $j < 4$ par $j < 3$.

Il est quand même étonnant que l'erreur n'ait pas été corrigée depuis le début, car cette partie du programme doit dater des premiers jours...

MA VIARIANTE – COUP EN ARRIERE

Afin de rendre le jeu un peu plus facile, je souhaite mettre au point un système de **retour en arrière**. Le principe sera que, lorsque l'on appuie sur Espace, le plateau redevient le même qu'au coup précédent. Cette possibilité ne peut se faire qu'une fois d'affilé.

Afin d'éviter l'abus de cette touche, le **score sera baissé de 5%**. Cela n'aura pas de grosse influence sur les petits scores, mais un score de 5000 passerait à 4750.

Pour se faire, je vais utiliser la Struct que j'ai mis en place un peu plus tôt. Je vais y insérer un nouveau tableau 4x4 que j'appellerai Back. Lorsque l'on appuiera sur Espace, alors notre tableau jeu.plateau prendra la valeur de jeu.back.

Autrement dit, il faut que jeu.back ait toujours la valeur du plateau précédent.

Pour cela, il va falloir, à chaque fois que notre plateau change, que jeu.back prenne la valeur de jeu.plateau, puis que jeu.plateau soit modifié et retourné.

Changements à effectuer

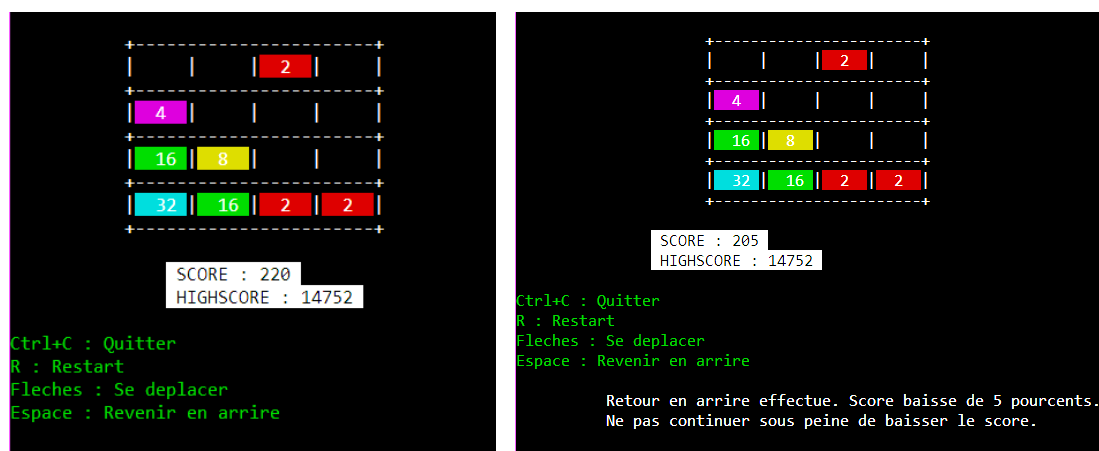
- A nouveau, il faudra adapter les commentaires qui devront descendre d'une ligne. Pour cela, on lance un "Rechercher et Remplacer".

- Il faudra trouver l'emplacement adéquat où jeu.back prend la valeur de jeu.plateau.

- Il faut dire au jeu qu'Espace n'est plus une Commande invalide, et la paramétrer pour lancer le Back. L'entier getch() correspondant à Espace est 32 (voir documentation annexe sur Ncurses).

Comme d'habitude, on effectuera tous les changements dans une branche parallèle de GIT. J'en profite pour changer la couleur des commandes afin d'y voir un peu plus clair.

Screen du jeu à un moment quelconque. Puis un mouvement vers le bas est effectué, et un Back est demandé. Screen à droite après avoir joué et appuyé sur espace.



MA VARIANTE – DOUBLAGE ALEATOIRE : PRINCIPE

Voici l'idée de ma variante, on va insérer 3 nouveautés.

- Des cases baissées aléatoirement.
- Des cases augmentées aléatoirement
- 1% de chance d'obtenir la suppression d'une case

Le dernier point est le plus complexe, il sera ajouté en dernier.

Les valeurs suivantes sont encore au stade de l'imagination. Elles sont susceptibles d'être modifiées par la suite.

EDIT : En effet, les probabilités ont été diminuées afin de réduire l'apparition de la fonction. Voir page 46.

Baisse aléatoire :

Il y a environ **10% de chance par coup** de subir une baisse aléatoire. On devrait donc la subir en moyenne 1 fois tous les 10 coups (en réalité ce sera moins que cela).

La fonction prendra une case aléatoire du tableau, si celle-ci valait :

- plus de 64**, alors elle est **divisée par 2**
- 64 ou 32**, alors elle passe à **8**
- moins de 32**, elle est **supprimée**
- nulle**, aucun changement

Le fait que cela puisse tomber sur une case nulle baisse la probabilité de 10%. Car si le tableau est en moyenne à moitié vide, alors la probabilité de baisser la valeur d'une case serait de 5%.

Augmentation aléatoire :

L'augmentation ne peut avoir lieu que si la baisse n'a pas eu lieu.

Sa probabilité est de **15% par coup**. De même qu'avant, cela peut tomber sur une case vide et n'avoir aucun effet. De même, la fonction prendra une case aléatoire du tableau, et si celle-ci valait :

- nulle ou 1024**, aucun changement
- moins de 32**, elle passe à **32**
- **32 ou 64**, elle passe à **128**
- de **128 à 512**, elle **double**

Commentaires associés

Dès que la fonction est appelée, elle produit un commentaire sur les lignes réservées 21 et 22. Elle indique **quelle case** a été visée, si c'est une baisse, une suppression, une augmentation, ou s'il n'y a aucun changement.

MA VARIANTE – DOUBLAGE ALEATOIRE : CODAGE

Codage de Baisse (et augmentation dans la foulée)

On commence d'abord par la fonction de **Baisse**. La baisse ou augmentation aura lieu après un coup. On va donc la placer dans le fichier main, dans la fonction "jouerUnCoup" à la suite de "commandeExecuter".

La baisse a 10% de chance de se produire. Donc la première chose à faire est de déterminer si l'évènement a lieu. S'il a lieu, on choisit une case aléatoire parmi les 16 du tableau. On récupère la valeur de cette case, et on fait 4 tests différents pour les 4 cas évoqués page précédente. On agit sur la valeur de cette case en fonction de sa valeur d'origine.

On retourne ensuite le plateau de jeu.

Afin de ne pas alourdir la mémoire inutilement, on va faire une fonction qui ne prend que le plateau du jeu, et pas le jeuGlobal. Je n'écris pas ma fonction à la suite mais parmi les fonctions qui gèrent l'aléatoire.

J'insère un premier test dans ma fonction main :

```
//Tests pour la fonction baisseOuAug à suppr une fois la fonc finie
for (int i=0 ; i<4 ; i++) {
    mvprintw(19,15,"Choisi un entier aléatoire entre 0 et 3 compris : %d" ,
    rand() % 3);
    getch(); }
```

Il faut remplace "% 3" par "% 4" pour obtenir des entiers de 0 à 3 **compris**.

La fonction ressemble à un enchainement de if , un peu comme la fonction afficheCase.

Une fois la fonction écrite

Finalement, j'ai décidé d'écrire les 2 parties (baisse et augmentation) dans la foulée. Après tout, ce sont 2 parties d'une même fonction, donc il est plus logique de l'écrire d'une seule traite.

Quelle bonne surprise de retrouver mon **Erreur de segmentation (core dumped) !**

Je désactive donc toutes les actions de ma fonction, place des printw pour voir jusqu'où la fonction s'exécute, et augmente les probabilités de 15% à 1/3 chacune pour ne pas à devoir attendre pour qu'elle s'exécute.

Je trouve rapidement mon erreur : lorsque la fonction n'entraîne ni baisse ni augmentation dans le tableau, alors les deux If était survolé, et la fonction n'avait pas de return. Mes 2 return étaient placés dans les If ou Else. Je sors donc les return et en place un unique à la fin.

Quelques corrections :

J'ai rajouté "valeur !=0" dans les if "valeur<32" sinon lorsqu'une case valait 0, cela actionnait à la fois 0 et inférieur à 32. J'ai supprimé les accents, qui ne sont pas supporté par Ncurses, et compléter quelques lignes oubliées.

De plus, la fonction se lance même lorsqu'un mouvement impossible est demandé... Cela ne devrait pas être le cas. Pour corriger cela, on lance la fonction baisseOuAug directement depuis l'intérieur des fonctions de mouvements. C'est le seul moyen de pouvoir contourner les mouvements impossibles. On a donc notre fonction qui peut se lancer depuis 4 endroits (les 4 mouvements différents) depuis le fichier annexe et non plus le fichier main.

Tests

Afin de tester ma variante, je lance la fonction plein de fois et pour observer si cela fonctionne et si l'aléatoire est respecté. Pour tester les cases à grandes valeurs, il faudrait démarrer avec un plateau qui en contient plusieurs.

A priori, tout fonctionne correctement. Commit sur GIT et Merge vers branche master.

Aperçu du code :

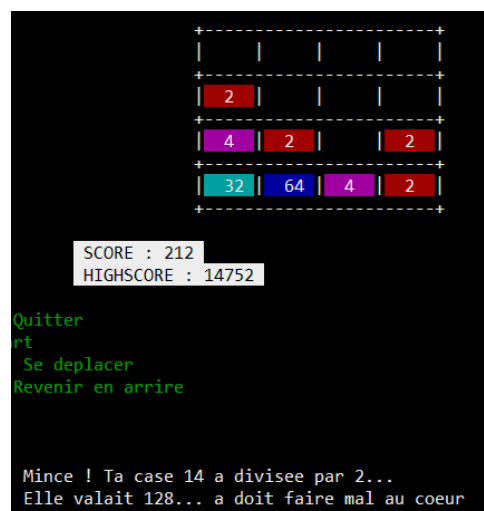
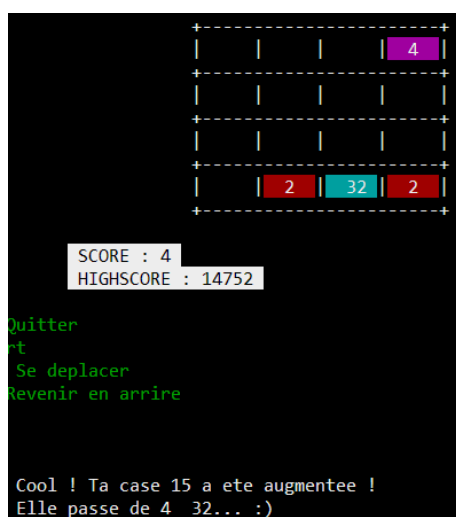
```
Matrice baisseOuAug(Matrice plateau) {
    // --- BAISSSE ---
    if( eventProbalsurN(3) ) { //10% de chance de se produire

        // Choix et coordonnées de la case
        int cord_x = rand() % 4;
        int cord_y = rand() % 4;
        int numCase = cord_x * 4 + cord_y + 1;
        int valeur = plateau[cord_x][cord_y];

        if (valeur == 0) { //
            mvprintw(21,10,"Quelle chance ! Ta case %d aurait du baisser...", numCase);
            mvprintw(22,10,"Mais elle est déjà vide. Tu y échappes pour cette fois.");
        }
        if (valeur < 32 and valeur!=0) { //
            plateau[cord_x][cord_y] = 0;
            mvprintw(21,10,"Mince ! Ta case %d a été supprimée...", numCase);
            mvprintw(22,10,"C'est pas trop grave, elle valait %d",valeur);
        }
        if (valeur == 32 or valeur == 64) {
            plateau[cord_x][cord_y] = 8;
            mvprintw(21,10,"Aie aie aie ! Ta case %d vient de baisser...", numCase);
            mvprintw(22,10,"Elle passe de %d à 8 :",valeur);
        }
        if (valeur > 64) {
            plateau[cord_x][cord_y] = valeur / 2;
            mvprintw(21,10,"Mince ! Ta case %d a été divisée par 2...", numCase);
            mvprintw(22,10,"Elle valait %d... ça doit faire mal au cœur",valeur);
        }
    } else {

```

ScreenShots de situations :



Des probabilités trop fortes

Avec ces probabilités de 10% et 15% (soit $1/10$ et $1/6.5$), les augmentations ou baisses arrivent trop fréquemment. Ils doivent être des événements plus rares qu'une fois sur 4. Je baisse donc à **4% pour la baisse, et 6% pour l'augmentation**. Ainsi, devrait rencontrer notre fonction une fois **tous les 6 ou 7 coups**.

4% revient à $1/20$, et 6% revient à $1/17$.

Des couleurs pour plus de lisibilité

Lire les messages à chaque fois est assez fatigant, il faudrait pouvoir distinguer directement si c'est une bonne nouvelle ou pas, sans avoir à lire et comprendre le message. Je vais donc mettre les commentaires positifs en fond Vert, les mauvaises nouvelles en fond rouge, et les neutres en fond blancs.

MA VARIANTE – SUPPRESSION

Je souhaite que **tous les 40 coups** (valeur susceptible de changer), on ait la **possibilité de supprimer une case** du jeu (ou non). Cela peut permettre de débloquer un jeu lorsque cela devient compliqué.

Fonctionnement, interaction avec le joueur

La struct jeuGlobal nous permet d'implanter aisément des fonctionnalités supplémentaires. Je vais ajouter à cette structure un compteur "int compteur40" qui sera initialisé à 0 dans la fonction plateauInitial. Après chaque coup joué (sauf mouvement impossible ou commande inconnue), le compteur sera incrémenté. Lorsqu'il atteindra 40, cela fera donc 40 coups joués.

Lorsque le jeu reconnaît que cela fait 40, il lance alors une fonction qui propose de supprimer une case en demandant son numéro (c'est un système plus simple pour l'utilisateur). Elle précisera que la case 1 est celle du coin Haut Gauche et 4 est le coin Haut Droite. Elle précisera que si l'utilisateur rentre 0, aucune case n'est supprimée.

Il faudra donc qu'elle vérifie que la valeur x rentrée respecte : $0 \leq x \leq 16$

Si la valeur de x n'est pas dans cet intervalle, elle est redemandée.

Codage de la suppression

Il faudra bien-sûr ajouter le compteur40 dans la struct de jeuGlobal. Il faudra également ajouter des incrémentations de ce compteur dans la fonction mouvementExecuter.

Il nous faudra créer une nouvelle fonction annexe, que je vais placer dans jouerUnCoup, dans mon fichier main. La fonction contiendra une boucle while qui demande le numéro de la case. Il faudra convertir ce numéro en coordonnées, et supprimer la case demandée. Puis afficher un message comme quoi cette case a été supprimée.

Changement dans la façon de prendre les coordonnées

Ncurses ne permet pas de demander un entier à l'utilisateur. On va donc lui demander le **numéro de la case en 2 chiffres** (01, 02, ...).

Si la première touche est un 0, alors ncurses nous renvoie 48, pour 1 il renvoie 49. On appelle c1 notre premier entier. **Si c1=48, numCase = c2-48.** Par exemple, 08 donnera (56-48) = 8.

Si c1=49, numCase = c2 - 38. Donc 16 donnera (56-38) = 16.

Conditions pour que le numéro soit correcte :

-c1 = 48 ou 49 sinon erreur

--numCase > -1 et numCase < 17 sinon erreur

Si on est dans un cas d'erreur, on redemande le numéro.

Conversion en coordonnées :

Si numCase = 0, on ne supprime rien.

Si numCase est un multiple de 4, soit (numCase%4=0) alors la case fait partie de la dernière colonne. Si le reste vaut 1, alors c'est la première colonne. On va décaler ce nombre pour faciliter la tâche. **numCase devient numCase - 1.** Dans ce cas, numCase%4=0 signifie première colonne, et un reste qui vaut 1 indique la deuxième colonne. Ça se traduit donc par :

y = numCase%4 et donc

x = numCase/4

plateau[x][y] = 0;

Un dernier changement dans le fonctionnement

Il n'est pas agréable que la possibilité de suppression arrive tous les 40 coups sans forcément qu'on l'ai choisi. Je vais donc mettre en place la possibilité de déclencher une suppression (avec la touche E). Ce sera donc un évènement qui se produit à un moment voulu par le joueur. Il ne sera disponible que tous les 70 coups. **Une nouvelle ligne de commande "E : Supprimer une case" sera insérée dans les commandes disponibles. Elle sera colorée en Rouge tant que ce n'est pas disponible, puis vert quand les 70 coups auront été atteints.** Par la suite, je vais finir mon jeu en changeant les touches, afin de tout rendre accessible facilement avec une seule main.

Je vais également baisser encore une fois les probabilités d'Augmentation et Baisse de 6% et 4% à 4% et 2%.

CHANGEMENTS DES TOUCHES

Le jeu est maintenant opérationnel et le rapport (annexe sur Ncurses compris) frôle les 60 pages. Il me reste une petite modification à faire sur les touches.

J'avais paramétré le jeu afin que ce soit les flèches qui enclenchent les mouvements, mais à présent, j'ai rajouté des fonctionnalités qui n'étaient pas prévu initialement. Il s'agit du Restart, du Retour en arrière, et de la suppression de case. Celles-ci sont déclenchées avec les touches **R, Espace et E**. Or il se trouve que ces 3 touches sont facilement accessibles de la main gauche si on remet en place les touches initiales de déplacements (ZQSD). Je vais donc à nouveau **changer les touches et remplacer les flèches par ZQSD**.

Interprétation des touches via getch() sur Ncurses :
Z : 122, Q : 113, S : 115, D : 100

Je remplace ces nombres dans 2 fonctions, celle qui vérifie que la commande est bien une commande connue (celle qui renvoie le "commande invalide"), ainsi que dans la fonction "commandeExecuter" qui agit en fonction de la touche pressée.

Le jeu est maintenant terminé ☺