IUT de Paris

PROJET DEMINEUR

Comparaison d'approche algorithmique

Yannis DA MOURA MONTEIRO 108
Mahydine LAZZOULI 111

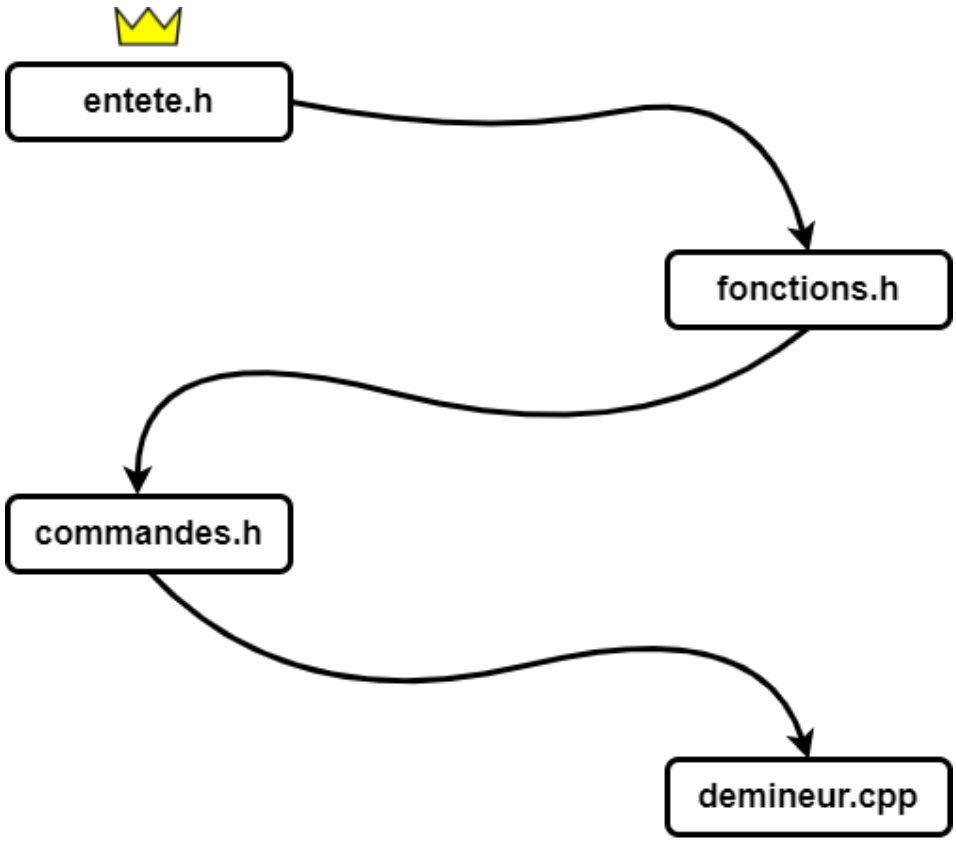
SOMMAIRE

INTRODUCTION

Le but du projet est de reproduire le jeu du démineur en l'adaptant aux attentes et aux règles du sujet, le tout en langage C++ avec 5 commandes élémentaires :

- Produire un problème
- Produire une grille
- Déterminer si la partie est gagnée
- Déterminer si la partie est perdue
- Produire un nouveau coup à partir d'une grille

GRAPHE DE DEPENDANCE



Ce graphe de dépendance nous explique de manière assez visuelle comment S'organise notre projet, le fichier d'entête « entete.h » est le cœur du programme, il n'est dépendant de personne et contient toutes les structures et définitions de type, ainsi que les briefs de chaque fonction.

Le fichier « fonction.h » lui, contient le code de toutes les fonctions et est dépendant d'entete.h car ses fonctions utilisent les structures de ce dernier.

« commande.h » est le header qui structure les commandes principales, Chaque commande (sauf la première) utilise plusieurs micro-fonctions, qui, unie, permettent le bon déroulement du processus.

Et enfin, « demineur.cpp » contient la fonction main, qui va permettre la détection et l'exécution des commandes.

JEUX D'ESSAI

Pour ces jeux d'essais nous avons utiliser les in et out fournis.

Commande 1 :

in1.txt

out1moi.txt

Commande 2 :

In2a.txt

out2a.txt

outmoi2a.txt

Le out attendu (à gauche) n'est pas le même que nôtre out, nous n'avons pas réussi à régler ce souci de propagation des cases vides malheureusement.

Commande 3 :

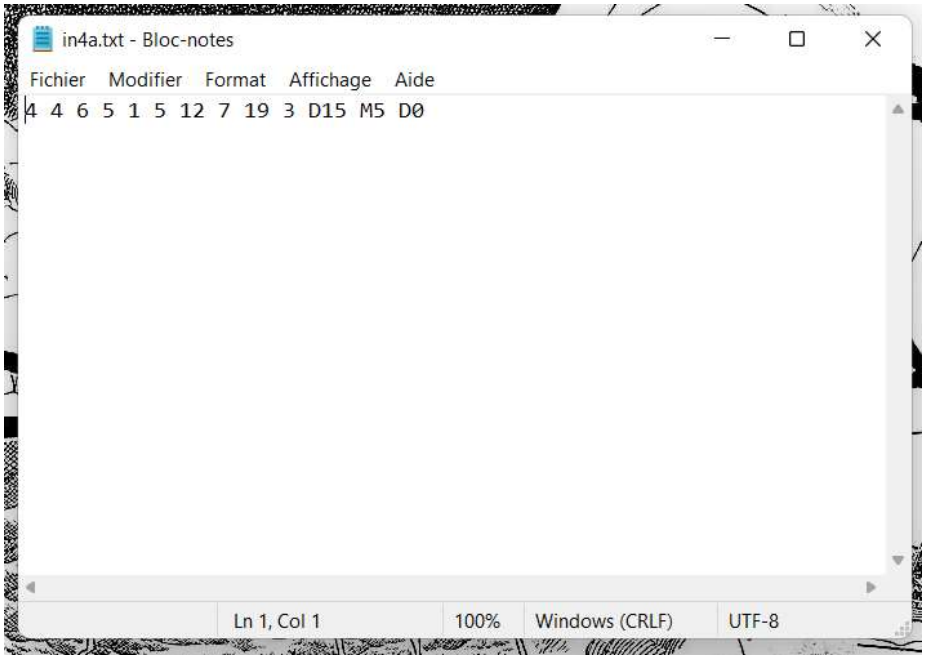
In3a.txt

out3amoi.txt

out3a.txt

Commande 4 :

In4a.txt

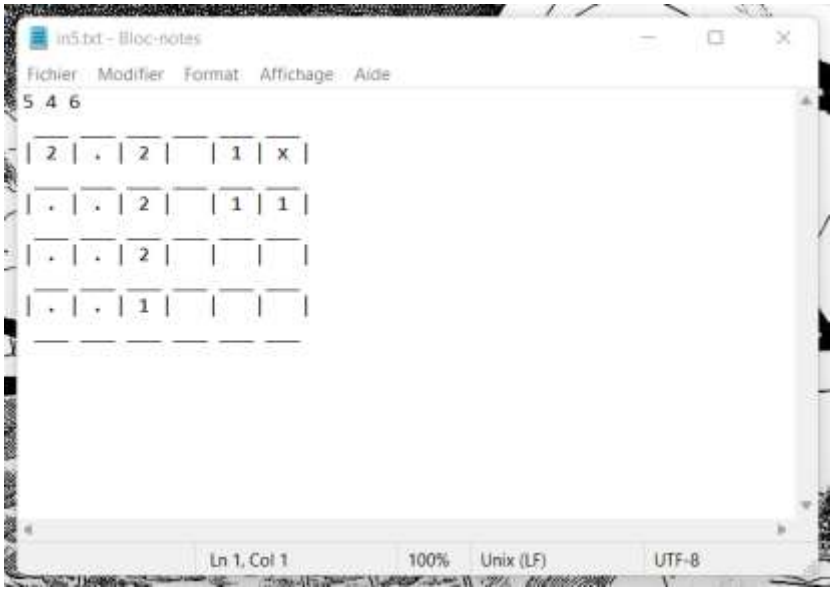


out4amoi.txt

out4a.txt

Commande 5 :

In5.txt



out5moi.txt

out5.txt

BILAN DU PROJET

Au départ, on a perdu pas mal de temps car on voulait faire une grosse fonction pour chaque commande, ce qui aurait été possible, certes, mais absolument pas optimisé, car en effets, plusieurs commandes ont besoin de faire les mêmes choses et ça n'aurait pas été optimal de simplement copier-coller leurs code, ont as donc, après réflexion, opté pour plusieurs petites fonctions qui exécutent chacune des petites taches, que l'on combinera dans une seule fonction par commande, selon les besoins de cette dernière.

Les difficultés rencontrées :

- L'allocation dynamique à été assez dure à assimiler.
- La propagation des cases vides pour crée les délimitations de numéro nous as aussi posé beaucoup de problème.

Ce qui est réussi :

- Nous sommes plutôt fier de la manière d'avoir organiser notre code, c'est très intuitif et logique, ça nous a fortement facilité le travail.
- Nous avons énormément appris en logique de code et en connaissances pure du C++ ce qui est, tout l'intérêt du projet.

CODE SOURCE

demineur.cpp :

```
#include "commandes.h"

int main() {

    Grille g;
    Item** item;

    unsigned int Commande = 0, lignes = 0, colonnes = 0, nombre_de_mines = 0;

    cin >> Commande;
        assert(Commande <= 5 && Commande >= 1 ); // Verifie que le numéro de
commande entré est entre 1 et 5
    cin >> lignes;
        assert(lignes >= 2); // Vérifie un minimum de 2 ou + lignes
    cin >> colonnes;
        assert(colonnes >= 2); // Vérifie un minimum de 2 ou + colonnes
    cin >> nombre_de_mines;
        assert(nombre_de_mines >= 1); // Vérifie un minimum de 1 ou + mines

    if (Commande == 1) Commande1(lignes, colonnes, nombre_de_mines);
    else if (Commande == 2) Commande2(g, item, lignes, colonnes,
nombre_de_mines);
    else if (Commande == 3) Commande3(g, lignes, colonnes, nombre_de_mines);
    else if (Commande == 4) Commande4(g, lignes, colonnes, nombre_de_mines);
    else if (Commande == 5) Commande5(g, item, lignes, colonnes,
nombre_de_mines);

    return 0;
}
```

commandes.cpp :

```cpp
#pragma one
#include "fonctions.h"

void Commande2(Grille& g, Item**item, unsigned int lignes, unsigned int colonnes, unsigned int nombre_de_mines){
	initialisation_jeu(g, lignes, colonnes, nombre_de_mines);
	triage_mines(g);
	triage_historique(g);
	initialisation_grille(&item, g);
	placage_des_mines(g,item);
	effectuer_coup(g,item);
	cases_adjacentes(g,item);
	affichage_grille(g,item);
}
void Commande3(Grille& g, unsigned int lignes, unsigned int colonnes, unsigned int nombre_de_mines){
	initialisation_jeu(g, lignes, colonnes, nombre_de_mines);
	is_won(g);
}
void Commande4(Grille& g, unsigned int lignes, unsigned int colonnes, unsigned int nombre_de_mines){
	initialisation_jeu(g, lignes, colonnes, nombre_de_mines);
	is_lost(g);
}
void Commande5(Grille& g, Item**item, unsigned int lignes, unsigned int colonnes, unsigned int nombre_de_mines){
	initialisation_jeu(g, lignes, colonnes, nombre_de_mines);
	triage_mines(g);
	triage_historique(g);
	initialisation_grille(&item, g);
	placage_des_mines(g,item);
	effectuer_coup(g,item);
	cases_adjacentes(g,item);
	affichage_grille(g,item);
	nouveau_coup(g);
	triage_historique(g);
	initialisation_grille(&item, g);
	placage_des_mines(g,item);
	effectuer_coup(g,item);
	cases_adjacentes(g,item);
	affichage_grille(g,item);
}
```

fonctions.cpp :

Commande 1 (problème) :

```cpp
#pragma once
#include"entete.h"

void Commande1(unsigned int lignes, unsigned int colonnes, unsigned int nombre_de_mines) {

    srand(time(NULL)); // initialise la fonction srand sur le temps actuel.

    // Calcul des dimensions de la grille pour pouvoir choisir des positions aléatoires sans débordé en dehors
    unsigned int dimensions = lignes * colonnes;

    // Affiche le nombre de lignes, de colonnes et de mines.
    cout << lignes << " " << colonnes << " " << nombre_de_mines << " ";

    // Boucle affichant une position aléatoire dans l'intervale 0:Dimensions
    // autant de fois qu'il y a de mines.
    for(int i = 0 ; i < nombre_de_mines ; i++){
        cout << rand() % dimensions << " ";
    }
}
```

Initialisation du jeu :

```cpp
void initialisation_jeu(Grille& g, unsigned int lignes, unsigned int colonnes, unsigned int nombre_de_mines) {

    // Entre les informations de lignes, colonnes, mines, dans la structure Grille
    g.lignes = lignes;
    g.colonnes = colonnes;
    g.nombre_de_mines = nombre_de_mines;

    // Choix et placement des mines stockées dans le tableau positionMines
    for (int i = 0 ; i < nombre_de_mines ; ++i) cin >> g.positionMines[i];

    // Stock le nombre de coups a effectuer souhaiter
    cin >> g.nombre_de_coups;
    assert(g.nombre_de_coups <= (lignes* colonnes));

    // Enregistrer les coups effectués autant de fois qu'il y a de coups a faire (nombre_de_coups)
    for (int m = 0 ; m < g.nombre_de_coups ; ++m) {
        cin >> g.coups[m]; // Stock l'origine du coup (marquage ou démasquage de case)
        cin >> g.positionCoups[m]; // Stock la position a laquelle effectué le coup (numero de la case)
    }
}
```

Triage des mines :

```
void triage_mines(Grille& g) {

    unsigned int temp;
    // Boucle s'executant autant de fois qu'il y a de mines
    for (int i = 0; i < g.nombre_de_mines; i++) {
        // Boucle s'executant autant de fois qu'il y a de mines²
        for (int j = 0; j < g.nombre_de_mines; j++) {
            // Interverti les valeurs si la première est inferieure a la deuxieme
            if (g.positionMines[i] < g.positionMines[j]) {
                temp = g.positionMines[i];
                g.positionMines[i] = g.positionMines[j];
                g.positionMines[j] = temp;
            }
        }
    }

}
```

Triage de l'historique de coups :

```
void triage_historique(Grille& g) {

    unsigned int temp = 0, temp_lettre = 0;

    for (int i = 0; i < g.nombre_de_coups; ++i) { // Boucle s'executant autant
de fois qu'il y a de coups
        for (int j = 0; j < g.nombre_de_coups; ++j) { // Boucle s'executant
autant de fois qu'il y a de coups²
            if (g.positionCoups[i] < g.positionCoups[j]) { // si la première
position du coup est inferieure a la suivante
                // Inverse la première position avec la deuxieme
                temp = g.positionCoups[i];
                g.positionCoups[i] = g.positionCoups[j];
                g.positionCoups[j] = temp;
                // Inverse la première lettre du coups avec la deuxieme
                temp_lettre = g.coups[i];
                g.coups[i] = g.coups[j];
                g.coups[j] = temp_lettre;
            }
        }
    }
}
```

Initialisation de la grille avec des cases masquées :

```
void initialisation_grille(Item*** it, Grille& g) {

    // Crée un tableau de caractère en allocation dynamique
    *it = new Item * [g.lignes];

    // Boucle qui se répète autant de fois qu'il y a de colonnes
    for (int x = 0 ; x < g.colonnes; ++x) {
        (*it)[x] = new Item[g.colonnes];
    }

    // Boucle qui se répète autant de fois qu'il y a de lignes
    for (int x = 0; x < g.lignes; x++) {
        // Boucle qui se répète autant de fois qu'il y a de colonnes
        for (int y = 0; y < g.colonnes; y++) {
            // Masque toute les cases de la grille
            (*it)[x][y] = '.';
        }
    }
}
```

Placement des mines dans la grille :

```
void placage_des_mines(Grille& g, Item** it) {

    unsigned int compteur = 0; // variable qui va servir de compteur
    unsigned int position = 0; // variable qui va servir d'indicateur de position

    // Boucle s'executant autant de fois qu'il y a de lignes
    for (int x = 0 ; x < g.lignes; ++x) {
        // Boucle s'executant autant de fois qu'il y a de colonnes
        for (int y = 0; y < g.colonnes; ++y) {
            // Si le numéro de case de la mine à l'indice compteur correspond a celui de la
position alors
            if (g.positionMines[compteur] == position) {
                // marquer la case d'un 'm' aux positions en deux dimensions : (x,y) sur la
grille
                it[x][y] = MINE;
                // incrémenté le compteur pour vérifier le reste des cases
                compteur++;
                // incrémenté la position pour vérifier le reste des cases
                position++;
            }
            // Sinon, la position actuelle sur la grille ne contient pas de mines donc..
            else {
                // incrémenté la position pour vérifier le reste des cases
                position++;
            }
        }
    }
}
```

Fait le coup a la position souhaitée :

```
void effectuer_coup(Grille& g, Item** it) {

    unsigned int compteur = 0; // variable qui va servir de compteur
    unsigned int position = 0; // variable qui va servir d'indicateur de position

    // Boucle s'executant autant de fois qu'il y a de lignes
    for (int x = 0; x < g.lignes; x++) {
        // Boucle s'executant autant de fois qu'il y a de colonnes
        for (int y = 0; y < g.colonnes; y++) {
            // Si le caractère du coup effectué à l'indice du compteur correspond a M
            // ET que la position du coup effectué à l'indice du compteur correspond a la
position sur laquelle on vérifie
            if (g.coups[compteur] == MARQUER && g.positionCoups[compteur] == position) {
```

```
                // effectue le coup sur la case aux positions en deux dimensions : (x,y)
sur la grille
                it[x][y] = MARQUE;
                // incrémenté la position pour vérifier le reste des cases
                position++;
                // incrémenté le compteur pour vérifier le reste des cases
                compteur++;
            }
            // Sinon si le caractère du coup effectué à l'indice du compteur correspond a D
            // ET que la position du coup effectué à l'indice du compteur correspond a la
position sur laquelle on vérifie
            else if (g.coups[compteur] == DEMASQUER && g.positionCoups[compteur] ==
position) {
                // Si la case est masquée
                if (it[x][y] == MASQUE) {
                    // on l'a démasque
                    it[x][y] = VIDE;
                    // incrémenté la position pour vérifier le reste des cases
                    position++;
                    // incrémenté le compteur pour vérifier le reste des cases
                    compteur++;
                }
            }
            // Sinon la position actuelle ne subit aucun coups de la part du joueur donc
            else {
                // incrémenté la position pour vérifier le reste des cases
                position++;
            }
        }
    }
}
```

Démasque les cases adjacentes :

```
void cases_adjacentes(Grille& g, Item** it) {

    unsigned int zero = 48; // 48 correspond au zéro en ASCII, servira en tant
que caractère

    // Boucle qui s'execute autant de fois qu'il y a de lignes
    for (int l = 0; l < g.lignes; ++l) {
        // Boucle qui s'execute autant de fois qu'il y a de colonnes
        for (int c = 0; c < g.colonnes; ++c) {
            // Si cet emplacement correspond a une mine, placer une mine
            if (it[l][c] == MINE) {
                it[l][c] = MINE;
```

```
                // Si a la fois, "l" est compris entre 1 et le nombre de
lignes
                // ET "c" est compris entre 1 et le nombre de colonnes
                if (l > 0 && l < g.lignes && c > 0 && c < g.colonnes) {
                    if (it[l - 1][c + 1] != MINE) it[l - 1][c + 1] = zero;
                    if (it[l - 1][c - 1] != MINE) it[l - 1][c - 1] = zero;
                    if (it[l - 1][c] != MINE) it[l - 1][c] = zero;
                    if (it[l][c - 1] != MINE) it[l][c - 1] = zero;
                    if (it[l][c + 1] != MINE) it[l][c + 1] = zero;
                    if (it[l + 1][c - 1] != MINE) it[l + 1][c - 1] = zero;
                    if (it[l + 1][c] != MINE) it[l + 1][c] = zero;
                    if (it[l + 1][c + 1] != MINE) it[l + 1][c + 1] = zero;
                }

                if (l == 0 && c == 0) {
                    if (it[l + 1][c + 1] != MINE) it[l + 1][c + 1] = zero;
                    if (it[l][c + 1] != MINE) it[l][c + 1] = zero;
                    if (it[l + 1][c] != MINE) it[l + 1][c] = zero;
                }

                if (l == 0 && c > 0 && c < g.colonnes) {
                    if (it[l + 1][c + 1] != MINE) it[l + 1][c + 1] = zero;
                    if (it[l][c + 1] != MINE) it[l][c + 1] = zero;
                    if (it[l + 1][c] != MINE) it[l + 1][c] = zero;
                    if (it[l + 1][c - 1] != MINE) it[l + 1][c - 1] = zero;
                    if (it[l][c - 1] != MINE) it[l][c - 1] = zero;
                }

                if (l == 0 && c == g.colonnes) {
                    if (it[l][c - 1] != MINE) it[l][c - 1] = zero;
                    if (it[l + 1][c] != MINE) it[l + 1][c] = zero;
                    if (it[l - 1][c + 1] != MINE) it[l][c + 1] = zero;
                }

                if (l > 0 && l < g.lignes && c == 0) {
                    if (it[l - 1][c] != MINE) it[l - 1][c] = zero;
                    if (it[l - 1][c + 1] != MINE) it[l - 1][c + 1] = zero;
                    if (it[l][c + 1] != MINE) it[l][c + 1] = zero;
                    if (it[l + 1][c + 1] != MINE) it[l + 1][c + 1] = zero;
                    if (it[l + 1][c] != MINE) it[l + 1][c] = zero;
                }

                if (l == g.lignes && c == 0) {
                    if (it[l - 1][c] != MINE) it[l - 1][c] = zero;
                    if (it[l - 1][c + 1] != MINE) it[l - 1][c + 1] = zero;
                    if (it[l][c + 1] != MINE) it[l][c + 1] = zero;
                }

                if (l == g.lignes && c > 0 && c < g.colonnes) {
```

```
                    if (it[l][c - 1] != MINE) it[l][c - 1] = zero;
                    if (it[l - 1][c - 1] != MINE) it[l - 1][c - 1] = zero;
                    if (it[l - 1][c] != MINE) it[l - 1][c] = zero;
                    if (it[l - 1][c + 1] != MINE) it[l - 1][c + 1] = zero;
                    if (it[l + 1][c + 1] != MINE) it[l + 1][c + 1] = zero;
                }

                if (l == g.lignes && c == g.colonnes) {
                    if (it[l][c - 1] != MINE) it[l][c - 1] = zero;
                    if (it[l - 1][c - 1] != MINE) it[l - 1][c - 1] = zero;
                    if (it[l - 1][c] != MINE) it[l - 1][c] = zero;
                }

                if (c == g.colonnes && l > 0 && l < g.lignes) {
                    if (it[l - 1][c] != MINE) it[l - 1][c] = zero;
                    if (it[l - 1][c - 1] != MINE) it[l - 1][c - 1] = zero;
                    if (it[l][c - 1] != MINE) it[l][c - 1] = zero;
                    if (it[l + 1][c - 1] != MINE) it[l + 1][c - 1] = zero;
                    if (it[l + 1][c] != MINE) it[l + 1][c] = zero;
                }
            }
        }
    }

    for (int l = 0; l < g.lignes; ++l) {
        for (int c = 0; c < g.colonnes; ++c) {
            if (it[l][c] == MINE) {
                it[l][c] = MINE;
                if (l >= 1 && l < g.lignes && c >= 1 && c < g.colonnes) {
                    if (it[l - 1][c + 1] != MINE) it[l - 1][c + 1] += 1;
                    if (it[l - 1][c - 1] != MINE) it[l - 1][c - 1] += 1;
                    if (it[l - 1][c] != MINE) it[l - 1][c] += 1;
                    if (it[l][c - 1] != MINE) it[l][c - 1] += 1;
                    if (it[l][c + 1] != MINE) it[l][c + 1] += 1;
                    if (it[l + 1][c - 1] != MINE) it[l + 1][c - 1] += 1;
                    if (it[l + 1][c] != MINE) it[l + 1][c] += 1;
                    if (it[l + 1][c + 1] != MINE) it[l + 1][c + 1] += 1;
                }

                if (l == 0 && c == 0) {
                    if (it[l + 1][c + 1] != MINE) it[l + 1][c + 1] += 1;
                    if (it[l][c + 1] != MINE) it[l][c + 1] += 1;
                    if (it[l + 1][c] != MINE) it[l + 1][c] += 1;
                }

                if (l == 0 && c > 0 && c < g.colonnes) {
                    if (it[l + 1][c + 1] != MINE) it[l + 1][c + 1] += 1;
                    if (it[l][c + 1] != MINE) it[l][c + 1] += 1;
                    if (it[l + 1][c] != MINE) it[l + 1][c] += 1;
```

```
                    if (it[l + 1][c - 1] != MINE) it[l + 1][c - 1] += 1;
                    if (it[l][c - 1] != MINE) it[l][c - 1] += 1;
                }

                if (l == 0 && c == g.colonnes) {
                    if (it[l][c - 1] != MINE) it[l][c - 1] += 1;
                    if (it[l + 1][c] != MINE) it[l + 1][c] += 1;
                    if (it[l - 1][c + 1] != MINE) it[l][c + 1] += 1;
                }

                if (l > 0 && l < g.lignes && c == 0) {
                    if (it[l - 1][c] != MINE) it[l - 1][c] += 1;
                    if (it[l - 1][c + 1] != MINE) it[l - 1][c + 1] += 1;
                    if (it[l][c + 1] != MINE) it[l][c + 1] += 1;
                    if (it[l + 1][c + 1] != MINE) it[l + 1][c + 1] += 1;
                    if (it[l + 1][c] != MINE) it[l + 1][c] += 1;
                }

                if (l == g.lignes && c == 0) {
                    if (it[l - 1][c] != MINE) it[l - 1][c] += 1;
                    if (it[l - 1][c + 1] != MINE) it[l - 1][c + 1] += 1;
                    if (it[l][c + 1] != MINE) it[l][c + 1] += 1;
                }

                if (l == g.lignes && c > 0 && c < g.colonnes) {
                    if (it[l][c - 1] != MINE) it[l][c - 1] += 1;
                    if (it[l - 1][c - 1] != MINE) it[l - 1][c - 1] += 1;
                    if (it[l - 1][c] != MINE) it[l - 1][c] += 1;
                    if (it[l - 1][c + 1] != MINE) it[l - 1][c + 1] += 1;
                    if (it[l + 1][c + 1] != MINE) it[l + 1][c + 1] += 1;
                }

                if (l == g.lignes && c == g.colonnes) {
                    if (it[l][c - 1] != MINE) it[l][c - 1] += 1;
                    if (it[l - 1][c - 1] != MINE) it[l - 1][c - 1] += 1;
                    if (it[l - 1][c] != MINE) it[l - 1][c] += 1;
                }

                if (c == g.colonnes && l > 0 && l < g.lignes) {
                    if (it[l - 1][c] != MINE) it[l - 1][c] += 1;
                    if (it[l - 1][c - 1] != MINE) it[l - 1][c - 1] += 1;
                    if (it[l][c - 1] != MINE) it[l][c - 1] += 1;
                    if (it[l + 1][c - 1] != MINE) it[l + 1][c - 1] += 1;
                    if (it[l + 1][c] != MINE) it[l + 1][c] += 1;
                }
            }
        }
    }
}
```

Affiche la grille, ses contours etc.. :

```
void affichage_grille(Grille& g, Item** it) {

    cout << g.lignes << " " << g.colonnes << endl;

    for (int k = 0 ; k < g.colonnes ; k++){ // Crée la première délimitation
de ligne, celle qui fait le toit de la grille
        cout << " ---";
    }

    cout << endl;

    for(int i = 0; i < g.lignes ; i++){ // Boucle Principale
        // Boucle ligne de mine et de séparations
        for(int j = 0; j < g.colonnes ; j++){
            cout << "| " << it[i][j] << " ";
        }
        // barre fermant la dernière case de chaque ligne
        cout << "|" << endl;
        // Boucle de délimitations lignes
        for (int x = 0 ; x < g.colonnes ; x++){
            cout << " ---";
        }
        // Saute une ligne après les delimitations
        cout << endl;
    }
}
```

Vérifie si la partie est gagnée ou non :

```
void is_won(Grille& g) {

    bool win = false; // Booleen si la partie est gagnée (true) ou non (false)
    unsigned int dimensions = g.lignes * g.colonnes;

    if (g.nombre_de_coups != dimensions - g.nombre_de_mines) win = false;
    else {
        // Boucle s'executant autant de fois qu'il y a de mines
        for (int i = 0; i < g.nombre_de_mines; ++i) {
            // Boucle s'executant autant de fois qu'il y a de mines²
            for (int j = 0; j < g.nombre_de_mines; ++j) {
                // Si aucun coup ne démasque une mine
                if (g.coups[i] == DEMASQUER && g.positionCoups[i] !=
g.positionMines[j]) win = true;
                // Sinon si un coup démasque une mine
                else win = false;
            }
        }
    }
    // Si win vaut false alors la partie est perdue
    if (win == false) cout << "game not won";
    // Sinon la partie est gagnée
    else cout << "game won";
}
```

Vérifie si la partie est perdue ou non :

```
void is_lost(Grille& g) {

    bool win = false; // Booleen si la partie est gagnée (true) ou non (false)

    // Boucle s'executant autant de fois qu'il y a de mines
    for (int i = 0 ; i < g.nombre_de_mines; i++) {
        if (g.coups[g.nombre_de_coups - 1] == MARQUER &&
g.positionCoups[g.nombre_de_coups - 1] != g.positionMines[i]) win = false; //
game lost
        if (g.coups[g.nombre_de_coups - 1] == DEMASQUER &&
g.positionCoups[g.nombre_de_coups - 1] == g.positionMines[i]) win = false; //
game lost
        if (g.coups[g.nombre_de_coups - 1] == DEMASQUER &&
g.positionCoups[g.nombre_de_coups - 1] != g.positionMines[i]) win = true; //
game not lost
        if (g.coups[g.nombre_de_coups - 1] == MARQUER && g.positionCoups[i] ==
g.positionMines[i]) win = true; // game not lost
    }
```

```
    // Si win vaut false alors la partie est perdue
    if (win == 0) cout << "game lost" << endl;
    // Sinon la partie est gagnée
    else cout << "game not lost" << endl;
}
```

Effectue un nouveau coup :

```
void nouveau_coup(Grille& g) {

    char coup;
    int position_coup;

    cin >> coup >> position_coup;
    cout << coup << position_coup << endl;

    // Ajoute le coup effectué dans la liste, une case après la dernière
    g.coups[g.nombre_de_coups + 1] = coup;
    // Ajoute la position du coup effectué dans la liste, une case après la
dernière
    g.positionCoups[g.nombre_de_coups + 1] = position_coup;
    // Incrémente le nombre de coups total de 1
    g.nombre_de_coups++;

    // Affiche la position qui vien d'être ajoutée
    cout << "la position que vous avez choisit est : " << position_coup <<
endl;
}
```

entete.cpp :

Includes et Structures :

```cpp
#include<iostream>  // Bibliothèque input output stream permettant la gestion du flux d'entrée sortie
#include<ctime>     // Bibliothèque ctime permettant de generer des nombres aléatoires
#include<cassert>   // Bibliothèque cassert pour verifier des conditions et stopper le programme si elle ne sont pas respectée

using namespace std; // permet de ne pas avoir à écrire "std::" à chaque utilisation d'une fonction iostream

#define TAILLE 100

typedef char Item;   // Le nouveau type "Item" correspond a un caractère unique et permettra une meilleure lisibilitée du code

enum Coup { MARQUER = 'M', DEMASQUER = 'D' }; // Enumération des coups possible d'une case dans une structure "Case"

enum Case { MINE = 'm', VIDE = ' ', MASQUE = '.', MARQUE = 'x'}; // Enumération des état possible d'une case dans une structure "Case"


// Structure Grille qui va contenir les informations relative a la grille
struct Grille {
    unsigned int lignes = 0;
    unsigned int colonnes = 0;
    unsigned int nombre_de_mines = 0;
    unsigned int nombre_de_coups = 0;
    int positionMines[TAILLE]; // Tableau d'entier qui contient la position de chaque mine
    int positionCoups[TAILLE];  // Tableau d'entier qui contient le/les numéro.s des cases que l'on marque ou démasque
    char coups[TAILLE]; // Tableau de caractère qui contient les lettres des coups que l'on souhaite faire, M ou D
};
```

Brief des fonctions :

```
/*
* @brief Renvoie une liste d'entier correspondand respectivement aux nombre de lignes, de colonnnes, de mines et génère aléatoirement leurs position
* @param [in] lignes, colonnes, nombre de mines
* @param [out] nombre de ligne, de colonne et de mine ainsi que leurs positions générée
*/
void Commande1(unsigned int lignes, unsigned int colonnes, unsigned int nombre_de_mines);


/*
* @brief Initialise la structure Grille avec le nombre de lignes, de colonnes, de mines et leurs positions
* @param [in] Grille& g, lignes, colonnes, nombre de mines
*/
void initialisation_jeu(Grille& g, unsigned int lignes, unsigned int colonnes, unsigned int nombre_de_mines);


/*
* @brief Trie les coups dans l'ordre croissant par rapport a la case visée.
* @param [in] Grille& g
*/
void triage_historique(Grille& g);


/*
* @brief Crée la base de la grille en initialisant une matrice
* @param [in] Item** it, Grille& g
*/
void initialisation_grille(Item*** it, Grille& g);


/*
* @brief Trie les mines dans l'ordre du plus petit au plus grand
* @param [in] Grille& g
*/
void triage_mines(Grille& g);


/*
* @brief Place les mines dans la grille de jeu
* @param [in] Grille& g, Structure Item
*/
void placage_des_mines(Grille& g, Item** it);


/*
```

```
* @brief Fonction permettant de marquer ou demasquer une case
* @param [in] Grille& g, Strucutre Item
*/
void effectuer_coup(Grille& g, Item** it);


/*
* @brief Affiche autour d'une case les informations des cases adjacentes
* @param [in] Grille& g, Structure Item
*/
void cases_adjacentes(Grille& g, Item** it);


/*
* @brief Affiche la grille, les lignes les colonnes avec les séparations ect..
* @param [in/out] Grille&, structure item
* @param [out] Grille visuelle
*/
void affichage_grille(Grille& g, Item** it);


/*
* @brief Determine si une partie est perdue ou pas
* @param [in] Structure grille
* return "game lost" ou "game not lost"
*/
void is_lost(Grille& g);


/*
* @brief Determine si une partie est gagnée ou pas
* @param [in] Structure grille
* @return "game won" ou "game not won"
*/
void is_won(Grille& g);


/*
* @brief Sert, comme son nom l'indique, a générer un nouveau coup
* @param [in] Grille& g, Strcture Item
*/
void nouveau_coup(Grille& g);
```