Sorbonne Université **M1 Mathématiques**
2022–2023 **4M056 : Programmation en C++**

TP noté 1 : jeu de la vie

Le but de ce TP est de programmer une implémentation du *jeu de la vie*. Ce "jeu" consiste en l'évolution d'un ensemble de cellules placées sur un quadrillage. Plus précisément, on dispose d'un quadrillage fini composé de cases vides et de cases occupées par une cellule. L'occupation des cases évolue en temps discret selon les règles suivantes :

- L'état (vide, ou occupée par une cellule) de chaque case au prochain instant est déduit de l'état de la case à l'instant présent et du nombre de cases actuellement occupées parmi les 8 cases voisines ;
- Si une case vide est voisine d'*exactement* trois cases occupées, alors une cellule apparaît sur cette case au tour suivant ;
- Si une cellule est voisine d'*une cellule ou moins* ou bien de *quatre cellules ou plus*, alors la cellule disparaît au tour suivant ;
- Toutes les autres cases gardent leur état au tour suivant.

Ces règles sont illustrées ci-dessous sur un exemple :

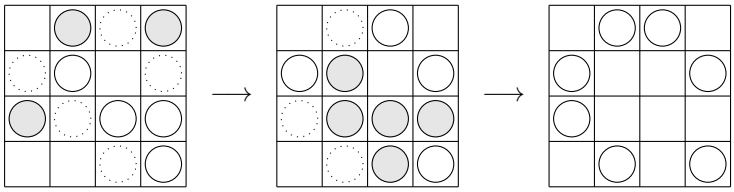

FIGURE 1 – Deux itérations du jeu de la vie. Les cases vides dans lesquelles une cellule va apparaître au tour suivant (cases ayant trois voisins) sont indiquées en pointillés. Les cellules qui vont mourir (n'ayant pas deux ou trois voisins) sont grisées.

On va représenter les configurations possibles du jeu par une classe `jeu_vie`. Le début du code de cette classe est le suivant :

```
class jeu_vie {
    private:
        int H, L;
        std::vector<std::vector<bool> > config;
    public:
        jeu_vie(int _H, int _L);
};
```

Les variables `H` et `L` correspondront à la taille du quadrillage considéré (hauteur et largeur). La variable `config`, qui est un vecteur de vecteurs de booléens contiendra la disposition des cellules sur la grille. Plus précisément, le booléen `config[i][j]` vaudra `true` si la case de la `i`-ème ligne de la `j`-ème colonne est occupée et vaudra `false` dans le cas contraire.

1. Dans un fichier `jeu_vie.hpp`, écrire la déclaration de la classe `jeu_vie`.

2. Écrire le code du constructeur pour la classe `jeu_vie` prenant deux arguments correspondant aux valeurs de `H` et `L`, donnant la bonne taille au vecteur `config` et le remplissant de valeurs `false` (autrement dit, on commence avec une grille vide).

3. Ajouter à la classe `jeu_vie` deux fonctions `hauteur` et `largeur` renvoyant respectivement les valeurs de `H` et `L`.

4. Écrire une méthode de prototype

```
int voisins(int i, int j) const;
```

renvoyant le nombre de cases occupées par une cellule qui sont voisines de la case sur la `i`-ème ligne et la `j`-ème colonne.

5. Tester la classe `jeu_vie` dans un fichier `test_jeu_vie.cpp` : créer un jeu de la vie `J` de taille `4x4` et afficher le nombre de voisins de la case `(0,0)`.

6. Déclarer une méthode `iteration` et la définir dans un fichier `jeu_vie.cpp`. Cette méthode effectue une itération du jeu de la vie : après l'appel de cette fonction, la variable `config` est modifiée de sorte à contenir la disposition des cellules à l'instant suivant.

7. Écrire une méthode `affichage` de prototype

```
void affichage(std::ostream &flux) const;
```

permettant d'écrire la configuration dans un flux sous la forme suivante :

```
4 4
.X.X
.X..
X.XX
...X
```

où sur la première ligne sont affichées la hauteur et la largeur de la grille.
Cet exemple correspond à l'état initial de la figure 1.

8. Écrire un constructeur de la classe `jeu_vie` qui prend en argument un flux d'entrée `std::istream&` et qui crée un `jeu_vie` à partir d'un fichier avec le format de la question précédente. Nous rappelons que si `s` est de type `std::string`, `s[j]` renvoie le `j`-ème caractère de `s`. Tester ce constructeur dans le `main` en créant un jeu de la vie `J` à partir du fichier `planeur.dat` dont le contenu est le suivant :

```
8 8
..X.....
X.X.....
.XX.....
........
........
........
........
........
```

9. Afficher les 20 premiers états successifs de `J`. Vous devriez obtenir une figure qui se déplace vers le bas à droite jusqu'à atteindre le coin opposé.

On rappelle que si `gen` est une variable de type `std::mt19937` et si `X` est une variable de type `std::bernoulli_distribution` initialisée avec la valeur `p`, alors l'appel de `X(gen)` renvoie une valeur aléatoire tirée selon la loi de Bernoulli de paramètre `p`. Ceci utilise la bibliothèque `random`, disponible en compilant avec l'option `-std=c++11`.

10. Écrire un constructeur de `jeu_vie` qui prend en argument deux entiers `H` et `L`, un paramètre `p` et une référence à un générateur `G` et remplit aléatoirement une grille de taille `H` × `L` avec des variables de Bernoulli de paramètre `p`.

11. Tester ce dernier constructeur dans le `main` avec `H=15`, `L=25` et `p=0.3`. Afficher la centième itération du jeu de la vie `J_random` partant de cette configuration.

12. Écrire une méthode `nb_cellules` qui renvoie le nombre de cellule occupées parmi les `H` × `L` cellules de la grille.

13. Calculer, pour chaque valeur p de $\{0.1, 0.2, \ldots, 0.9\}$, le nombre moyen de cellules occupées au bout de 100 itérations du jeu de la vie. Cette moyenne sera calculée en partant de 100 conditions initiales tirées aléatoirement de façon indépendante avec des lois de Bernoulli de paramètre p. On se placera encore sur une grille 15×25. Comment le résultat dépend-il de la valeur de p ?