

PROJET DE PROGRAMMATION AVANCÉE EN C++ LE JEUX DE LA VIE

Sebastien.Varrette@imag.fr (Bureau BR.4.07)

Volker.Muller@uni.lu (Bureau BC 3.01)

Il sera tenu compte dans la notation de ce projet de la qualité de la programmation et de la clarté du rapport attendu.

1 Introduction et cahier des charges

Le but de ce projet est l'écriture d'un programme permettant d'exécuter le célèbre "jeu de la vie". Ce projet est à réaliser seul. Un rapport (réalisé en \LaTeX) devra être remis en fin de projet accompagné des sources. Ce rapport aura pour objectif de décrire vos classes¹ et de répondre aux quelques questions apparaissant tout au long de cet énoncé. Une archive `tgz` contenant ces éléments sera déposée sur le serveur Moodle. Une introduction à l'édition de document sous \LaTeX est disponible à l'adresse http://www-id.imag.fr/~svarrett/tutoriaux.html#tutorial_LaTeX.

2 Le jeu de la vie

Ce jeu consiste à faire évoluer un ensemble de cellules réparties sur une grille rectangulaire découpée en cases. Chaque case contient une cellule qui peut être dans deux états : soit *active*, soit *inactive*. Chaque cellule possède 3, 5 ou 8 voisins.

A chaque cycle, l'état de toutes les cellules est évalué simultanément et elles évoluent comme suit :

Règle 1 : une cellule inactive s'active si elle est entourée d'**exactement 3** cellules actives ;

Règle 2 : une cellule active ne survit **que** si elle est entourée de 2 ou 3 cellules actives (et devient inactive sinon).

La figure 1 fournit un exemple d'évolution sur 2 cycles (la situation d'arrivée est stable). Pour déterminer l'état au cycle $i + 1$ à partir de l'état au cycle i , on

1. Des outils comme Doxygen <http://www.doxygen.org> pourraient vous être utiles

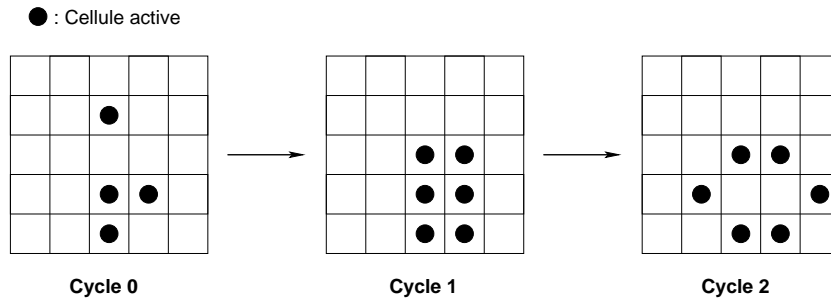


FIGURE 1 – Le jeu de la vie : exemple d'évolution sur 2 cycles.

procède comme suit :

1. Pour chaque cellule active au cycle i , déterminer celles qui survivent en appliquant la règle (2) ;
2. Pour chaque cellule inactive au cycle i , déterminer celles qui s'activent, **en utilisant la configuration du cycle i** en appliquant la règle (1).

Pour programmer ce problème, nous allons modéliser les cellules par des objets de la classe `Cell`. La grille de cellule sera représentée par un objet de la classe `Environment`. Enfin, une classe `GameOfLife` gèrera l'évolution de l'environnement dans le cadre du jeu de la vie.

Chaque définition de classe fait l'objet d'une partie de cet exercice. Bien entendu, si vous bloquez sur une question qui demande la réalisation d'une fonction, vous pouvez supposer cette fonction réalisée pour continuer l'exercice.

2.1 La classe `Cell`

Pour caractériser l'état d'une cellule, on utilisera l'énumération :

```
enum CELL_STATE {INACTIVE, ACTIVE};
```

La classe `Cell` possède un seul attribut privé, son état (champ `_state`). Le constructeur de cette classe place la cellule créée dans l'état `INACTIVE` par défaut.

De plus, la classe `Cell` implémente les méthodes suivantes :

- `bool isActive()` : (★) permet de savoir si une cellule est active ou non ;
- `void setActive()` : (★) permet d'activer une cellule ;
- `void setInactive()` : (★) permet de désactiver une cellule ;
- `void print()` : affiche un caractère caractérisant l'état de la cellule. Par convention, on utilisera le caractère dièse '#' pour afficher une cellule active et le caractère espace ' ' pour une cellule inactive.

1. Définir la classe `Cell`. On veillera à ne définir qu'un seul constructeur par défaut ainsi qu'un constructeur par copie. On utilisera dans les deux cas une liste d'initialisation.

IMPORTANT : Les méthodes marquées par un astérisque ★ seront définies directement dans la définition de la classe. Pour la méthode `print`,

on se contentera du prototype, son corps étant défini dans les questions suivantes.

2. Les méthodes définies directement dans une classe sont `inline`. Que signifie ce terme ? Quel est l'intérêt de cette approche ?
3. Dans l'optique de l'affichage, on souhaite définir les constantes globales `INACTIVE_CHAR` et `ACTIVE_CHAR` (correspondant aux caractères utilisés pour caractériser l'état d'une cellule selon qu'elle soit active ou pas).
 - (a) Définir ces deux constantes, en les initialisant aux valeurs choisies par convention.
 - (b) Définir la méthode `print` en utilisant ces deux constantes.

2.2 La classe `Environment`

Un objet de la classe `Environment` sera caractérisé par sa hauteur h , sa largeur l et un tableau de $h \times l$ éléments de types `Cell`.

On fournit la définition de la classe `Environment` :

```
class Environment {
    typedef Cell * CellLine; // redéfinition d'un type ligne de cellule
                               // propre à la classe Environment
    CellLine * _CellGrid;    // grille de cellule
    unsigned int _h;        // hauteur : nombre de lignes
    unsigned int _l;        // largeur : nombre de colonnes
public:
    /** Constructeur - Destructeur */
    Environment(unsigned int h = 5, unsigned int l = 5);
    Environment(const Environment & env); // constructeur par recopie
    ~Environment();
    /** Accesseur - mutateur */
    unsigned int getH() const { return _h; }
    unsigned int getL() const { return _l; }
    void setH(unsigned int h) { _h = h; }
    void setL(unsigned int l) { _l = l; }
    /** Autres Methodes */
    // récupère la cellule en position (x,y)
    Cell getCellAtPosition(unsigned int x, unsigned int y) const
        { return _CellGrid[y][x]; }
    // rend active/inactive la cellule en position (x,y)
    void setActiveAtPosition(unsigned int x, unsigned int y)
        { _CellGrid[y][x].setActive(); }
    void setInactiveAtPosition(unsigned int x, unsigned int y)
        { _CellGrid[y][x].setInactive(); }
    // permet de savoir si la cellule en position (x,y) est active ou non
    bool isActiveAtPosition(unsigned int x, unsigned int y)
        { return _CellGrid[y][x].isActive(); }
    // renvoie le nombre de cellules actives parmi les voisins de (x,y)
    short numberOfActiveNeighbours(unsigned int x, unsigned int y);
    // pour l'affichage
    void print();
};
```

1. Définir le constructeur par défaut de la classe `Environment`. Par convention, toutes les cellules de l'environnement seront inactives.
2. Définir le constructeur par copie de la classe `Environment`.
3. Définir le destructeur de la classe `Environment`.
4. Définir la méthode `numberOfActiveNeighbours` qui, pour une position (x,y) donnée sur la grille, renvoie le nombre de cellules actives parmi les voisines² de la cellule en (x,y).
5. Définir la méthode `print` qui affiche l'environnement et l'état des cellules qu'il contient.

2.3 La classe `GameOfLife`

La classe `GameOfLife` ne contient qu'un seul attribut privé : un environnement `_env`. Le constructeur de cette classe reçoit donc deux paramètres de type `unsigned int` : la largeur et la hauteur de cet environnement.

En outre, cette classe dispose des méthodes publiques suivantes :

- `void randomInit()` : initialise l'environnement ; l'état des cellules qui le compose est tiré aléatoirement.
- `void execute(unsigned int nbSteps)` : exécute le jeu de la vie pendant `nbSteps` cycles et affiche l'environnement après chaque cycle.

Enfin, la méthode privée `void _executeOneCycle()` exécute un seul cycle du jeu de la vie. Toutes les cellules évoluent simultanément.

1. Définir la classe `GameOfLife`. Pour toutes les méthodes, on se contentera de déclarer leur prototype sauf pour le constructeur dont le corps sera défini directement (à l'aide d'une liste d'initialisation).
2. Définir la méthode `randomInit`. On utilisera pour cela la librairie C `<stdlib.h>` qui fournit les fonctions `srand` (qui permet d'initialiser le générateur aléatoire) et `rand` (qui fournit une valeur aléatoire uniformément distribuée entre 0 et `RAND_MAX`).

On pourra initialiser le générateur aléatoire de la façon suivante :

```
#include <stdlib.h>
#include <time.h>
...
void GameOfLife::randomInit() {
    srand(time(NULL)); // initialisation du générateur aléatoire.
    ...
}
```

Ensuite, on peut remarquer que `(double)rand()/RAND_MAX` est un nombre réel compris entre 0 et 1. On pourra donc utiliser la fonction suivante qui renvoie un état de cellule aléatoire :

```
CELL_STATE getRandState() {
    return ((double)rand()/RAND_MAX < 0.5)?INACTIVE:ACTIVE;
}
```

3. Définir la méthode `_executeOneCycle`.

2. On rappelle que selon les valeurs de x et y, une cellule a 3, 5 ou 8 voisines.

4. Définir la méthode `execute`.
5. Tester votre programme! (en particulier, faites apparaître les tests effectués dans votre rapport).

FIN bravo!