UNIVERSITÉ DE MONTPELLIER
UNIVERSITÉ DE MONTPELLIER - FACULTÉ DES SCIENCES ET TECHNIQUES
PLACE EUGÈNE BATAILLON - 34095 MONTPELLIER CEDEX 5

Licence 2 informatique – 2018/2019

HLIN302 – Travaux Dirigés n° 3

Programmation impérative avancée

Alban MANCHERON et Pascal GIORGI

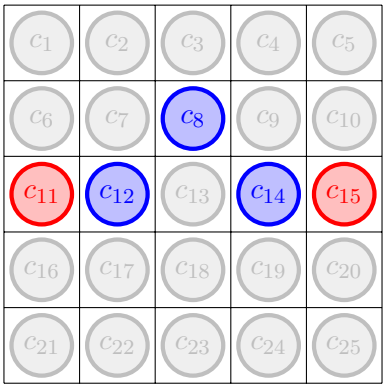
1 Le jeu de la vie (suite)

Rappel

En 1970, John Horton CONWAY imagine un automate cellulaire (*i.e.*, un modèle où chaque état conduit mécaniquement à l'état suivant à partir de règles prédéfinies) qu'il intitule « Jeu de la vie ». Il ne s'agit pas réellement d'un jeu vu qu'il n'y a pas d'interaction particulière avec un joueur, mais d'un modèle permettant d'observer des phénomènes dynamiques (oscillations, stabilité, saturation, ...).
Le concept est de définir un état d'origine où des objets élémentaires sont placés dans le plan, puis de calculer, à partir de règles simples si ces objets demeurent à leur place, s'ils disparaissent ou bien s'ils créent un nouvel objet dans le plan. Il s'agit donc bien d'un automate, puisque d'un état donné, on arrive à un nouvel état par application d'un calcul.
Cet automate est dit cellulaire car le plan est représenté par un grille à deux dimensions où chaque case (« cellule ») est dans un état choisi dans un ensemble fini (ici, soit la cellule est occupée, soit elle est vide). Il a été baptisé « jeu de la vie » par analogie entre l'état des cases (occupées ou vides) et les cellules biologiques (vivantes ou mortes) ainsi que par les mécanismes de reproduction.

Dans le sujet de TD-TP précédent, vous avez commencé à définir une classe `Cellule` permettant de représenter une cellule du jeu de la vie. De manière simpliste, le jeu de la vie consiste en une grille à deux dimensions où chaque case est occupée par une cellule. Chaque cellule évolue (meurt ou naît) au cours du temps en fonction de son voisinage. Afin de mieux représenter les états des cellules, nous avons proposé d'utiliser des couleurs. Le bleu correspond à une cellule qui est née, le vert à une cellule qui a survécu, le rouge à une cellule qui va mourir et le noir à une cellule morte. La couleur jaune peut être ajouté pour représenter une cellule qui vient juste de naître et qui va mourir. La figure ci-dessous illustre un état du jeu de la vie pour une grille 5×5 de cellules.

FIGURE 1 – Exemple d'une grille 5×5 de cellules



Une solution pour représenter une cellule est la classe suivante :

Déclaration de la classe Cellule

```
#ifndef __CELLULE_V4_H
#define __CELLULE_V4_H

#include <string>

class Cellule {
 public:

  enum Couleur {
    NOIR,
    BLEU,
    VERT,
    ROUGE,
    JAUNE,
    NB_COULEURS
  };

 private:
  size_t age;
  unsigned int x, y;
  Couleur couleur;

 public:

  // Constructeurs
  Cellule(); // morte par défaut
  Cellule(bool etat, unsigned int x, unsigned int y);

  // Accesseurs en lecture
  bool getVivante() const;
  unsigned int getX() const;
  unsigned int getY() const;
  Couleur getCouleur() const;

  // Accesseurs en écriture
  void setX(unsigned int x);
  void setY(unsigned int y);
  void setVivante(bool etat);

  // renvoie vrai si la cellule courante est vivante et est voisine de c
  bool estVoisine(const Cellule &c) const;
  // affiche la cellule
  void print() const;

  // spécifie qu'une cellule doit mourir au prochain tour du jeu (-> changement de couleur)
  void doitMourir();
};

// Renvoie vrai si la cellule est de la couleur passée en paramètre, faux sinon.
bool CelluleEstDeLaCouleur(const Cellule &cellule, Cellule::Couleur couleur);

// Retourne la chaîne correspondant à la couleur passée en paramètre
std::string Couleur2String(Cellule::Couleur c);

#endif
```

1.1 Vers une population de cellules

Afin de poursuivre, nous souhaitons maintenant représenter des grilles du jeu de la vie. Nous parlerons plutôt de N-population pour faire référence à une grille de $N \times N$ cellules.

1. Écrire un programme qui créé une 3-population qui a seulement des cellules mortes sur la diagonale et l'anti-diagonale de la grille.
2. Ajouter une méthode `print` à la classe **`Cellule`** qui affiche les coordonnées et la couleur (au format texte) d'une cellule.
3. Modifier votre programme pour que l'utilisateur puisse modifier interactivement l'état d'une cellule de la 3-population puis afficher la population modifiée à l'écran.
4. Quelles sont les possibilités pour interdire qu'un utilisateur modifie volontairement dans un programme l'état d'une cellule de la population ?

1.2 Une classe Population (encapsulation de cellules)

Afin de mieux gérer les N-population de cellules, nous nous intéressons maintenant à l'écriture d'une classe **`Population`**. Pour l'instant, la taille de notre population doit être définie comme une constante dans le code (vous devrez utiliser les macros du préprocesseur pour changer facilement cette constante, vous verrez plus tard comment faire cela plus proprement). L'objectif de cette classe est de ne pas fournir d'accès direct aux cellules de la population et d'avoir un minimum de contrôle d'accès aux données, à l'inverse de la version tableau vue à l'exercice précédent.

1. Écrire la signature d'une classe minimale **`Population`** permettant la création d'une N-population de cellules mortes, l'initialisation d'une configuration aléatoire avec k cellules vivantes et l'accès à des copies des cellules (pour l'instant pas d'autres méthodes).
2. Ajouter la signature des accesseurs en lecture des informations suivantes (nombre de cellules vivantes, mortes, qui sont nées ou qui vont mourir).
3. Afin de faciliter l'écriture du code de ses accesseurs, nous allons définir une méthode `nb_cellules` qui pour une couleur donnée en paramètre calcule le nombre de cellules de cette couleur dans la population. Dans quel mode de protection doit on placer cette méthode ? Donner le code complet de cette méthode et des accesseurs en lecture précédents.
4. Comment fournir une copie d'une cellule précise dans la population ? Donner la signature.
5. Est-il possible de renvoyer directement une cellule de la population sans la copier et en interdisant de la modifier ?
   (a) Proposer le code complet correspondant si vrai.
   (b) Est-ce que le code est différent de la version avec copie ? Expliquer pourquoi.
6. En considérant, la méthode **`void`** `printCell(size_t i, size_t j)` permettant d'afficher la cellule positionnée aux coordonnés (i, j). Proposez une solution permettant de mieux protéger cet accès en vérifiant la validité des coordonnées dans la grille. Si les coordonnées ne sont pas valides, le code terminera le programme en appelant **`std::`**`terminate()` disponible avec **`#include`** `<exception>`. Votre solution devra être réutilisable pour d'autres méthodes.
7. On s'intéresse maintenant à fournir une méthode permettant de générer la population suivante (application des règles du jeu de la vie). En considérant la signature de cette méthode :

```
1 Population next() const;
```

   (a) Doit-on ajouter d'autres méthodes à la classe **`Population`** ? Motiver vos réponses.
   (b) Donner le code complet de la méthode `next`.

1.3 Une autre approche de la classe Population

Vous aurez remarqué que la solution précédente pour la classe **`Population`** oblige à gérer les coordonnées de la grille à la fois dans les cellules et dans la population, ce qui n'est pas satisfaisant. Afin de palier à ce problème, nous allons gérer la population par une représentation dite « creuse » (on ne stocke que ce qui est pertinent).

L'idée est de ne stocker que les cellules vivantes de la population, les autres étant mortes par complémentarité. Par exemple, pour stocker la configuration donnée dans la Figure 1, il suffit de ne stocker que 5 cellules : $C_8, C_{11}, C_{12}, C_{14}, C_{15}$, au lieu de 25 dans l'approche précédente.

En pratique, les cellules vivantes seront stockées dans un tableau statique. Ceci pose un problème pour la taille du tableau que l'on ne peut pas connaître *apriori*. Une solution correcte [1] consiste à choisir une constante **nmax** bornant le nombre de cellules vivantes dans la population [2]. En stockant en plus de ce tableau le nombre de cellules vivantes, il est alors facile de connaître la partie utile du tableau. Cette classe doit se charger de créer des populations vides et permettre la naissance (ajout) et la mort (suppression) de cellules.

1. Comment doit-on définir l'unique constructeur de la classe **`PopulationVivante`**.
2. Donner la déclaration de la classe **`PopulationVivante`**.
3. Comment fournir une copie d'une cellule précise dans la population ? Donner le code de votre méthode et expliquer la différence avec celle de la classe **`Population`**.
4. Comment faire pour proposer une méthode `next`.

1.4 Quelles populations (Qui sera le vainqueur ?)

- Programmer sur machines les codes complets des deux classes **`Population`** et **`PopulationVivante`**.
- Proposer deux programmes permettant de tester vos deux classes et de voir leurs avantages et leurs inconvénients.

1. Ce n'est ni la seule ni la meilleure, mais elle reste simple.
2. Il est clair que **nmax**=N^2 est toujours valable mais n'est pas le plus économe en mémoire.