UNIVERSITÉ DE MONTPELLIER
UNIVERSITÉ DE MONTPELLIER - FACULTÉ DES SCIENCES ET TECHNIQUES
PLACE EUGÈNE BATAILLON - 34095 MONTPELLIER CEDEX 5
Licence 2 informatique – 2016/2017

HLIN302 – Travaux Dirigés n° 2

Programmation impérative avancée
Alban MANCHERON et Pascal GIORGI

1 Intervalles

http://fr.wikipedia.org/wiki/Intervalle_(mathématiques)

En mathématiques, un intervalle (du latin *intervallum*) est étymologiquement un ensemble compris entre deux valeurs. Cette notion première s'est ensuite développée jusqu'à aboutir aux définitions suivantes. [···] Initialement, on appelle intervalle réel un ensemble de nombres délimité par deux nombres réels constituant une borne inférieure et une borne supérieure. Un intervalle contient tous les nombres réels compris entre ces deux bornes.

1.1 Une classe simple

Afin de manipuler la notion d'intervalle dans un programme, on souhaite pouvoir :
— déclarer une variable de type intervalle, en l'initialisant ou non ;
— l'afficher sous la forme $[b_i, b_s]$ [1] ;
— connaître la valeur de chacune de ses bornes (« accesseurs en lecture ») ;
— modifier la valeur de chacune de ses bornes (« accesseurs en écriture ») ;
— connaître sa longueur ;
— étant donné un réel, savoir s'il appartient à l'intervalle.

À partir de ce cahier des charges :

1. Écrire la déclaration de la classe `Itv` (uniquement le `.h`).
2. Écrire un programme de test qui créé un ou deux intervalles et appelle les diverses méthodes.
3. Implémenter la classe (fichiers `.h` et `.cpp`) en `TP` ainsi que le programme de test.

1.2 Compléments

Étant donnés deux intervalles I_1, I_2 de la droite réelle, I_1 peut être :
— égal à I_2 ;
— inclus strictement dans I_2, *i.e.* inclus mais non égal (attention, une des bornes peut être commune : $[3, 5]$ est strictement inclus dans $[2, 5]$) ;
— disjoint de I_2 (aucun point commun) ;
— accolé à I_2 (la borne supérieure de l'un est égale à la borne inférieure de l'autre) ;
— imbriqué avec I_2 (le cas qui reste quand aucun des quatre précédents n'est vérifié).

À partir de ce complément au cahier des charges :

1. Écrire dans la classe `Itv` les méthodes qui renvoient un booléen pour chacun de ces cas.
2. Écrire une fonction et une méthode permettant de translater un intervalle d'une valeur réelle `x` passée en paramètre.
3. Écrire un programme de test.

1. b_i pour « borne inférieure », b_s pour « borne supérieure ».

2 Le jeu de la vie

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 ce TD et les suivants, nous utiliserons le « jeu de la vie » comme support afin d'illustrer l'ensemble des concepts vus en cours. Les solutions retenues ne seront pas nécessairement optimales, mais ont pour objectif de mettre en application les connaissances que vous devez acquérir.

2.1 Une simple cellule

Une cellule peut-être représentée par son état (vivante ou morte) et sa position dans le plan (ses coordonnées x et y sont figées). En outre, chaque cellule crée est initialisée avec ses valeurs propres (morte ou vivante, ainsi que sa position dans le plan). Enfin, une cellule doit permettre de tester si une cellule donnée est voisine et vivante (une cellule voisine morte ne nous intéresse pas).

1. Écrire la définition la plus simple possible de la classe **`Cellule`** correspondant à cette description.
2. Écrire une fonction `test_cell`, qui étant donnée une cellule permet d'afficher l'état de ses attributs sur la sortie standard.
3. Écrire un programme principal simple appelant fonction `test_cell` et la méthode `estVoisin` qui puisse générer une sortie similaire à

```
L'objet c1 est à l'adresse mémoire 0x7ffcca3a67c0
L'objet c2 est à l'adresse mémoire 0x7ffcca3a67d0
La cellule (à l'adresse mémoire 0x7ffcca3a67c0) = {vivante, 1x2}
La cellule (à l'adresse mémoire 0x7ffcca3a67d0) = {morte, 1x3}
La cellule c1 est voisine de c2.
La cellule c2 n'est pas voisine de c1.
```

4. Est-il judicieux de créer un constructeur par défaut pour cette classe ? Justifier.
5. Modifier la déclaration de la classe pour éviter une manipulation incontrôlée des attributs et permettre au code suivant de compiler.

```
1  int main(int argc, char** argv) {
2    Cellule c1(true, 1, 2), c2;
3    c2.setVivante(!c2.getVivante());
4    c2.setX(c1.getX());
5    c2.setY(c1.getX() + c1.getY());
6    PrintCell(c1); PrintCell(c2);
7    test_cell(c1); test_cell(c2);
8    PrintVoisines(c1, c2); PrintVoisines(c2, c1);
9    return 0;
10 }
```

   Quelles modifications cela implique-t-il dans l'implémentation précédentes ?

2.2 Une cellule en couleur

Dans le jeu de la vie, l'évolution des cellules est entièrement déterminée par ce qui l'entoure. En effet, une cellule morte possédant exactement trois cellules voisines vivantes ressuscite (c'est un jeu) et une cellule vivante qui a moins de deux ou plus de trois cellules voisines vivantes décède (elle meurt de solitude ou bien étouffée).

Afin de mieux visualiser le processus, les cellules sont généralement représentées colorées en fonction de leur évolution. Ainsi, les cellules mortes sont noires, celles qui naissent ou ressuscitent en bleu, les cellules en pleine force de l'âge en vert, les cellules qui vont mourir en rouge.

1. Proposer un mécanisme permettant de représenter cette information (sémantique) dans la classe.
2. Modifier la déclaration et l'implémentation de la classe **`Cellule`** pour intégrer cette information.
3. Écrire une fonction permettant de savoir si une cellule donnée est d'une couleur donnée.

1. Étant donnée une cellule c qui vient de naître et qui ne survivra pas au prochain tour, quelle doit être sa couleur ? Ce cas peut-il se produire ? Le cas échéant de telles cellules seront arbitrairement colorées en jaune. Que faut-il alors modifier pour que le code précédent reste valide ?
2. Que manque-t-il pour pouvoir assigner correctement les couleurs jaune et rouge ? Quelles solutions vous semblent pertinentes pour gérer ces situations ? La présence d'un attribut `couleur` est-elle indispensable ?

– Annexes –

A Rappels

Une classe s'écrit toujours dans deux fichiers : un fichier d'extension `.h` et un fichier d'extension `.cpp`. Le fichier `.h` contient la déclaration de la classe (attributs et signatures des méthodes) tandis que le fichier `.cpp` contient l'implémentation des méthodes.

De plus, la directive de pré-processeur **`#include`** ne doit servir à inclure que des fichiers de déclaration (`.h`), jamais de fichier d'implémentation (`.cpp`).

Enfin, l'existence des espaces de nom a pour objectif de limiter la définition de variables, de types, de classes et de fonctions au sein de ces espace. L'usage d'un espace de nom dans un fichier a pour seul et unique objectif d'alléger l'écriture du code. Il est par conséquent inconcevable – et donc lourdement sanctionné – de voir apparaître la ligne **`using namespace std;`** dans un fichier d'entête.

B L'énumération

En C++, l'énumération permet de créer un type associé à une liste de constantes entières. L'intérêt de ce type est de regrouper des constantes d'un même contexte dans un même type. La déclaration la plus basique d'une énumération est la suivante :

```
enum {
    CSTE_1,
    CSTE_2,
    ...,
    CSTE_n
};
```

Par défaut, la valeur d'une constante est la valeur de la constante précédente plus un, sachant que la première constante a pour valeur par défaut `0`.

Ainsi dans l'énumération ci-avant, `CSTE_1` vaut 0, `CSTE_2` vaut 1, ..., `CSTE_n` vaut $n-1$.

Il est également possible d'assigner des valeurs spécifiques à certaines de ces constantes en les initialisant lors de la déclaration :

```
enum {
    CSTE_1 = 1,
    CSTE_2,
    ...,
    CSTE_n,
    NB_CSTES = CSTE_n - CSTE_1 + 1
};
```

Le code précédent déclare `CSTE_1` valant 1, `CSTE_2` valant 2, ..., `CSTE_n` valant n et `NB_CSTES` valant n également.

Enfin, il est possible – et pratique – de nommer une énumération afin de pouvoir l'utiliser (*e.g.*, pour déclarer une variable du type de l'énumération) :

```
enum MonEnumeration {
    CSTE_1 = 1,
    CSTE_2,
    ...,
    CSTE_n,
    NB_CSTES = CSTE_n - CSTE_1 + 1
};
MonEnumeration ma_variable = CSTE_2;
```