

## Types de coordonnées et ensemble de coordonnées (préliminaire au projet)

Le code écrit dans ce TP sera utilisé dans le projet. Il est donc impératif de le tester de manière très poussée pour ne pas introduire de bug dans le projet.

Vous commencerez à travailler en binôme lors de cette séance, et vous commencerez également à utiliser `git` pour travailler en binôme.

Les méthodes des classes seront publiques, sauf précision du contraire. Par contre les attributs seront toujours privés.

Tapez la commande `./course.py fetch Projet` afin d'initialiser votre dépôt `git` pour le projet. Tous vos fichiers seront à créer dans le dossier ainsi obtenu.

► **Exercice 1. (Infrastructure et Makefile)** Dans cette partie vous allez mettre en place votre structure de fichiers et faire votre `Makefile`. Pour créer et gérer les nouvelles classes qui vous permettront de manipuler les coordonnées, vous aurez les fichiers suivants :

— <code>coord.cpp</code>	— <code>main.cpp</code>
— <code>coord.hpp</code>	— <code>Makefile</code>
— <code>test.cpp</code>	— <code>doctest.h</code>

Les fichiers `coord.hpp` et `coord.cpp` vous permettront de déclarer et décrire les classes et fonctions relatives aux coordonnées. Le fichier `test.cpp` vous permettra d'activer les tests dans un fichier à part (voir slide 10 du cours 5). Ce fichier de test ne contiendra que les lignes suivantes :

```
#define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
#include "doctest.h"
```

Vous écrirez les tests à côté des fonctions qu'ils testent dans les fichiers `.cpp`. Par exemple, le fichier `coord.cpp` commencera avec les lignes suivantes :

```
#include "doctest.h"
#include "coord.hpp"
```

Vous remarquerez qu'on ne spécifie pas de configuration. Le fichier `doctest.h` est inclut simplement pour que l'on puisse définir les `TEST_CASE` relatifs à chaque méthode.

Enfin, le fichier `main.cpp` ne contiendra pas de `TEST_CASE`, c'est là que vous programmerez la fonction `main` de votre projet pour lancer la simulation. Il commencera par les lignes suivantes.

```
#define DOCTEST_CONFIG_IMPLEMENT
#include "doctest.h"
```

Après les avoir compilés, il faudra lier les fichiers `.o` avec le fichier de test ou avec le fichier `main`. Ainsi, la commande décrivant la fabrication de l'exécutable de test dans le `Makefile` pourra être :

```
tests: test.o coord.o animal.o population.o grille.o jeu.o
———>g++ -o tests test.o coord.o animal.o population.o grille.o jeu.o
```

Au début du projet vous n'aurez pas tous ces `.o` mais seulement `test.o` et `coord.o`. La commande est donc plus courte, et sera complétée au fur et à mesure de votre projet.

Quand vous créez de nouveaux fichiers, **pensez, dès la création d'un fichier, à faire la commande** `git add nomDuFichier` pour que `git` le gère. Voici les opérations à faire :

1. Créez vos différents fichiers en n'oubliant pas les `#include`.
2. Ajoutez dans `git` vos fichiers avec `git add`.
3. Créez votre `Makefile` en n'oubliant pas de lier les différents fichiers entre eux.
4. Ajoutez votre `Makefile` dans `git` avec `git add`.
5. Faites un `prog-mod submit Projet MonGroupe` pour vérifier que tout va bien.
6. Si votre binôme est connu et présent vous pouvez vous reporter au document `AideGit.pdf` d'aide sur `Git` pour lui donner l'accès à votre travail (n'y passez pas plus de 10 minutes).

► **Exercice 2. (La classe Coord)** On cherche à manipuler des coordonnées dans une grille, c'est-à-dire la paire constituée d'un numéro de ligne et d'un numéro de colonne.

1. Dans `coord.hpp`, définir une constante `TAILLEGRILLE` qui décrira la taille de la grille (comme la grille sera carrée, `TAILLEGRILLE` est simplement un entier, on pourra prendre 40).
2. Créer la classe `Coord` selon la description ci-dessus. Rappel : Les attributs seront toujours privés.
3. Coder un constructeur de `Coord` qui prend en paramètre un numéro de ligne *lig* et un numéro de colonne *col*. Déclencher une exception si les coordonnées ne sont pas correctes.  
Dans ce constructeur, la taille de la grille est connue. Ainsi le constructeur de coordonnées **vérifie qu'on ne construit pas une coordonnée en dehors de la grille**. D'autre part, on ne met que des getters et pas de setters dans l'interface de coordonnées. Ainsi, il n'est pas possible de modifier une coordonnée. Il n'y a donc jamais de coordonnées incorrectes ce qui évitera les erreurs de segmentation causées par un dépassement des bornes du tableau. C'est aussi pour cette raison qu'il n'y a pas de constructeur par défaut de coordonnées.  
Rappel : les méthodes de vos classes doivent être publiques, sauf précision du contraire.  
Remarque : Réfléchir à ce qui doit être mis dans le fichier `.hpp` et ce qui doit être mis dans le fichier `.cpp`.
4. Créer un `TEST_CASE` dans `coord.cpp` avec seulement la création d'une coordonnée (sans `CHECK`) et **vérifiez dès maintenant que vous arrivez à compiler et exécutez le test**. Si vous avez des difficultés, c'est le moment de demander de l'aide à votre chargé de TP.
5. Écrire les getters (mais surtout pas les setters) permettant de récupérer les coordonnées d'une `Coord`.
6. Pour tester votre constructeur et vos getters, complétez le `TEST_CASE` précédent pour vérifier que le numéro de ligne et de colonne sont corrects. Rajoutez aussi des tests `CHECK_THROWS_AS` pour vérifier qu'une erreur est bien levée si les coordonnées sont en dehors de la grille.
7. Surcharger l'opérateur `<<` qui affiche les coordonnées sous la forme : *(lig, col)*.
8. Affichez une coordonnée dans la fonction `main` de `main.cpp` et vérifiez ainsi que vous arrivez bien à compiler le `main`.
9. Créez un `TEST_CASE` pour l'affichage dans `coord.cpp`.
10. Surcharger les opérateurs `==` et `!=` et créez les tests correspondants.

11. Dans la suite, on aura parfois besoin d'encoder une paire de coordonnées en un seul entier. Pour cela, on va mettre en correspondance la paire  $(lig, col)$  avec l'entier  $lig \times \text{TAILLEGRILLE} + col$ . Ceci revient à numérotter les cases de gauche à droite et de haut en bas, selon le sens usuel de lecture (dans le tableau ci-dessous, on abrège **TAILLEGRILLE** en  $t$ ).

		col				
		0	1	2	...	$t - 1$
lig	0	0	1	2	...	$t - 1$
	1	$t$	$t + 1$	$t + 2$	...	$2t - 1$
	2	$2t$	$2t + 1$	$2t + 2$	...	$3t - 1$
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
	$t - 1$	$t^2 - t$	$t^2 - t + 1$	$t^2 - t + 2$	...	$t^2 - 1$

Écrire un constructeur de **Coord** qui prend en paramètre un entier seul et crée la paire de coordonnées correspondantes, et une méthode **toInt** qui renvoie l'entier associé à une paire de coordonnées et les tests correspondant

12. Tester scrupuleusement cette première classe, en réfléchissant bien à tous les cas limites possibles, afin d'éviter qu'elle n'induisse des bugs dans votre projet.
13. Comme les coordonnées ne sont jamais modifiées, toutes les méthodes sont **const**, vérifiez que c'est bien le cas dans votre implantation.

### ► Exercice 3. (Ensembles)

Nous allons maintenant réaliser une structure pour représenter des ensembles d'entiers. Cette structure sera utilisée plusieurs fois dans le projet :

- parfois pour représenter des ensembles de coordonnées, grâce à l'encodage des paires de coordonnées en entiers seuls,
- parfois pour représenter des ensembles d'animaux, via l'identifiant unique de chacun.

Les deux principales opérations nécessaires sur ces ensembles seront l'ajout d'un nouvel entier, et le tirage au hasard de l'un des entiers. Remarque : étant donné l'usage qui sera fait de cette structure dans la suite, vous n'aurez pas besoin d'une vraie structure d'« ensemble » mathématique. Une structure de « sac » pourra suffire.

1. Créer une class **Ensemble** contenant un tableau d'entiers  $t$  de taille fixe **MAXCARD**, et un entier **card** donnant le nombre d'éléments présents dans l'ensemble. On considérera que seuls les éléments présents dans les **card** premières cases de  $t$  sont significatifs. Ci-dessous on note  $k$  la valeur de **card**. Le contenu à partir de l'indice  $k$  peut être arbitraire, il ne sera pas pris en compte.

0	1		$k - 1$	$k$		<b>MAXCARD</b>
$e_0$	$e_1$	...	$e_{k-1}$	?	...	?

Notez que la taille fixe **MAXCARD** représente également le cardinal maximal d'un ensemble. À vous de déterminer une valeur appropriée pour **MAXCARD**.

2. Surcharger l'opérateur **<<** pour **Ensemble**. N'afficher que les parties significatives du tableau sous-jacent (vous pouvez passer par une méthode intermédiaire **affiche** de la classe qui est appelée par la fonction **operator<<** comme on l'a vu dans le cours)
3. Écrire un constructeur par défaut de **Ensemble**, créant un ensemble vide, c'est-à-dire un ensemble de cardinal zéro et vérifiez que vous arrivez à l'afficher.
4. Écrire une méthode **estVide()** indiquant la vacuité d'un ensemble et **cardinal()** renvoyant son cardinal.

- Écrire la méthode `ajoute` qui prend en paramètre un objet et l'ajoute à l'ensemble, et la méthode `tire` qui renvoie un élément pris au hasard dans l'ensemble, et le retire. L'une et l'autre doivent déclencher une exception si l'ensemble ne permet pas l'opération : on ne peut pas ajouter un nouvel entier à un ensemble ayant déjà atteint la taille maximale, ni retirer un entier d'un ensemble vide. Notez que ces deux méthodes doivent préserver la propriété selon laquelle tous les éléments de l'ensemble sont consécutifs dans le tableau entre les indices 0 et `card`. Vous pourrez vouloir effectuer une ou plusieurs permutations pour assurer cela.

L'aléatoire peut être introduit dans l'une ou l'autre de ces deux méthodes au choix, mais il est inutile de le faire dans les deux : on peut ajouter les nouveaux éléments à la fin et tirer les éléments au hasard, ou insérer les éléments au hasard et tirer à la fin.

**Aide :** la fonction `rand()` (voir l'explication à la fin du TP2), renvoie un nombre aléatoire. Pour obtenir un nombre aléatoire entre 0 et  $n$  (compris), il faut donc appliquer un modulo  $(n + 1)$  au résultat de l'appel à `rand()`.

- Tester minutieusement toutes ces fonctions avec des `TEST_CASE`, car s'il y a une erreur, c'est beaucoup plus facile de la détecter à ce stade que quand elle aura comme effet de rendre incohérent le comportement d'un animal à l'écran.

#### ► Exercice 4. (Création de la grille)

- On s'intéresse maintenant à trouver les coordonnées des voisines d'une case dans une grille de taille `TAILLEGRILLE` lignes et `TAILLEGRILLE` colonnes. Par exemple, considérons la grille ci-dessous où `TAILLEGRILLE = 5` :

(0, 0)	(0, 1)	(0, 2)	(0, 3)	(0, 4)
(1, 0)	(1, 1)	(1, 2)	(1, 3)	(1, 4)
(2, 0)	(2, 1)	(2, 2)	(2, 3)	(2, 4)
(3, 0)	(3, 1)	(3, 2)	(3, 3)	(3, 4)
(4, 0)	(4, 1)	(4, 2)	(4, 3)	(4, 4)

- Les voisines de (2, 1) sont (1, 0), (1, 1), (1, 2), (2, 2), (3, 2), (3, 1), (3, 0), (2, 0).
- Les voisines de (3, 4) sont (2, 3), (2, 4), (3, 3), (4, 3), (4, 4).
- Les voisines de (0, 0) sont (0, 1), (1, 0), (1, 1).

Créer la méthode `voisines` de la classe `Coord` qui renvoie un `Ensemble` des voisines de la case considérée. Notez qu'un `Ensemble` ne peut stocker que des entiers : il faut passer par la méthode de codage `toInt`.

Si `lig` est la ligne de `c` et `col` est la colonne de `c`, on peut collecter les coordonnées des voisines de `c` dans un ensemble `ev` de la manière suivante :

```

pour i allant de  $i_{min}$  à  $i_{max}$  faire
  pour j allant de  $j_{min}$  à  $j_{max}$  faire
    si  $(i, j) \neq (lig, col)$  alors
      ev.ajoute(Coord{i, j}.toInt())
    finsi
  finpour
finpour

```

avec :

$$\begin{aligned}
 i_{min} &= \max(lig - 1, 0) \\
 i_{max} &= \min(lig + 1, TAILLEGRILLE - 1) \\
 j_{min} &= \max(col - 1, 0) \\
 j_{max} &= \min(col + 1, TAILLEGRILLE - 1)
 \end{aligned}$$

2. Tester sur plusieurs cas au milieu et sur les bords de la grille que votre fonction est correcte. Créer des TEST\_CASE en vérifiant par exemple la cardinalité ou l'affichage de l'ensemble obtenu.

**Attention !!! Cette fonction est cruciale pour le reste de votre projet !  
Il est impératif de la tester avec un maximum de cas limites possibles.**