université PARIS-SACLAY

Travaux Pratiques Projet Termites

Cours de programmation modulaire

—Licence MI/IM - Info 3—

Coordonnées, directions et grille pour le projet Termites

Nous allons réaliser les **types abstraits** suivants :

- `Coord` pour contenir les coordonnées d'un point de la grille ;
- `Direction` pour contenir une direction ;
- `Grille` pour contenir l'état du monde à un moment donné ;
- `Termite` pour coder un termite.

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

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

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. Pour information, notre corrigé fait environ 500 lignes de code (en comptant les commentaires). Pour les tests, on a en plus 200 lignes (que je considère insuffisantes).

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 ou dans des sous-dossiers.

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

- coord.cpp
- coord.hpp
- test.cpp
- Makefile
- doctest.h

Les fichiers `coord.hpp` et `coord.cpp` vous permettront de déclarer et décrire les classes et fonctions. Le fichier `test.cpp` vous permettra d'activer les tests dans un fichier à part (voir cours 5), ce qui permettra pour la suite de votre projet de faire la simulation sans forcément lancer les tests, et inversement.

Si toutes vos fonctions sont testées avec des cas de test `doctest`, ce fichier de test ne contiendra que deux lignes :

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

Si vous avez besoin d'écrire des procédures de test autres que des cas de test, le fichier `test.cpp` peut contenir un `main` qui lance à la fois les tests de `doctest` et vos procédures de test.

Vous écrirez les tests à côté des fonctions qu'ils testent dans les fichiers `.cpp`. Après les avoir compilés, il faudra lier les fichiers `.o` avec le fichier de test. Ainsi, la commande de fabrication de l'exécutable de test dans le `Makefile` pourra être :

```
tests: test.o coord.o termite.o grille.o jeu.o
————→g++ -o tests test.o coord.o termite.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 :

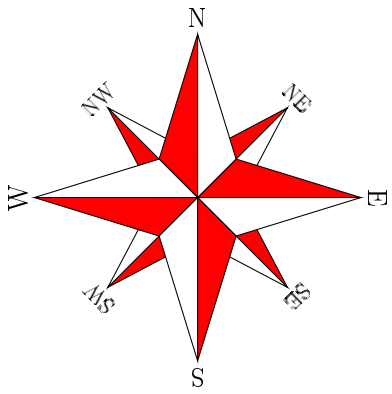
1. Créez vos différents fichiers en n'oubliant pas les `#include` et les gardes d'inclusions multiples dans les `.hpp`.
2. Ajoutez dans `git` vos fichiers avec `git add`.
3. Créez votre `Makefile` en n'oubliant pas de lier les différents fichiers entre eux.
4. Ajoutez votre `Makefile` dans `git` avec `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).

1 Les coordonnées

On veut réaliser le type abstrait **Coord** pour coder des coordonnées dans une grille, c'est-à-dire la paire constituée d'un numéro de ligne et d'un numéro de colonne dont on aura vérifié qu'ils ne sortent pas des limites fixées. Pour toutes les questions de la suite de ce TP, à vous de réfléchir dans quel fichier il faut les réaliser.

7. Définissez une constante `tailleGrille`
8. Créer la classe `Coord`.
9. Coder le constructeur de la classe `Coord` qui prend en paramètre un numéro de ligne lig et un numéro de colonne col, et retourne une nouvelle paire de type `Coord` initialisée à la coordonnée (lig, col) en levant une exception si les coordonnées ne sont pas dans la grille.
10. Coder les deux getter `getLig` et `getCol`, qui permettent respectivement de récupérer le numéro de ligne et le numéro de colonne d'une coordonnée.
11. Surcharger l'opérateur d'affichage pour les coordonnées en affichant la valeur sous la forme : (lig, col).
12. Tester tout ceci soigneusement en vérifiant tous les cas limites. On rappelle que les tests doivent être conservés.
13. Surcharger l'opérateur d'égalité retournant vrai si deux coordonnées sont égales.
14. Tester cet opérateur dans un nouveau cas de test.

2 Direction



Le type direction représente l'un des huits point cardinaux (nord-ouest, nord, nord-est, est, sud-est, sud, sud-ouest, ouest). On va étendre la bibliothèque `coord` pour qu'elle gère aussi les directions.

15. Coder le type enuméré `Direction`.
16. Surcharger l'opérateur d'affichage sur une `Direction`.
17. Coder les fonctions `aGauche` et `aDroite` qui prennent en paramètre une direction et retournent la direction située juste à sa gauche (respectivement droite).
18. Tester systématiquement ces fonctions. En particulier, vérifier qu'en partant de n'importe quelle direction :
    (a) si l'on tourne à gauche puis à droite, on doit être revenu dans la direction initiale ;
    (b) si l'on tourne à droite puis à gauche, on doit être revenu dans la direction initiale ;
    (c) si l'on tourne 8 fois à gauche, on doit être revenu dans la direction initiale ;
    (d) si l'on tourne 8 fois à droite, on doit être revenu dans la direction initiale ;
19. Coder maintenant la fonction `devantCoord` qui retourne la coordonnée devant une coordonnée donnée dans une direction donnée. Pour les cas où l'on est sur le bord de la grille, ne rien faire de particulier, on va laisser se propager l'exception levée par le constructeur.
20. Tester très soigneusement cette fonction. En particulier, vérifier que l'exception est correctement levée quand on est sur un bord ou dans un angle.
21. Tester systématiquement que, en partant d'une coordonnée dans n'importe quelle direction, si l'on avance, puis tourne 4 fois à droite, puis avance encore, on revient à la coordonnée de départ.

3 Grille

On s'intéresse maintenant au type abstrait `Grille`. On rappelle que la grille est un tableau à deux dimensions de taille `tailleGrille`. On pourra tester plusieurs tailles par la suite. Chaque case de la grille peut être vide, contenir une brindille ou un termite. Comme vu en TD, les termites seront rangés dans un tableau et chaque termite aura un numéro correspondant à sa position dans le tableau.

22. Créer les deux fichiers `grille.hpp` et `grille.cpp`, avec les directives `#include` correctes, et les ajouter convenablement dans le `Makefile`.
23. Coder la structure `Case`, indiquant si la case contient une brindille ou un termite (avec le numéro du termite si il y en a un, -1 sinon). C'est une structure dont les attributs sont publics et qui sera cachée de l'utilisateur par le type abstrait `Grille`. S'il n'y a ni termite ni brindille, la case est alors considérée comme vide.
24. Écrire un constructeur par défaut qui fabrique une case vide.
25. Coder ensuite le type abstrait `Grille` (qui est donc un tableau 2D de `Case`).
26. Coder les méthodes suivantes :

    ```
    void Grille::poseBrindille(Coord c);
    void Grille::enleveBrindille(Coord c);
    bool Grille::contientBrindille(Coord c) const;
    void Grille::poseTermite(Coord c, int idT);
    void Grille::enleveTermite(Coord c);
    int Grille::numéroTermite(Coord c) const;
    bool Grille::estVide(Coord c) const;
    ```

    Attention ! Pour que vous puissiez débogger plus facilement plus tard, les fonctions devront lever des exceptions quand elles ne peuvent pas faire le travail. Par exemple, la fonction `enleveTermite` ne peut pas enlever de termite s'il n'y en a pas dans la case. De la même manière `poseTermite` refusera de faire son travail s'il y a déjà un termite ou une brindille dans la case.

    De plus, pour chacune de ces fonctions, écrire un cas de test qui vérifie en profondeur le bon fonctionnement de chacune de ces méthodes.
27. Surcharger l'affichage de la grille ; dans un premier temps, on ne peut pas savoir dans quel sens regarde un termite. On affichera donc les termites par `T`. Quand on aura écrit la classe termite et initialisé correctement le jeu, il faudra reprendre ce code pour en faire une fonction ou méthode `afficheJeu`
28. Créer un fichier `projet.cpp` avec un `main` où l'on crée une grille avec quelques brindilles et termites et on l'affiche. Vérifier que tout marche bien.