université PARIS-SACLAY

Travaux Pratiques Info 121 n°9

Cours de programmation impérative

—Licence MPI L1 S2 - Info 121—

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.

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

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. Créez un nouveau fichier `coord.cpp`, puis dans ce fichier :

1. Coder le type structure `Coord`.
2. Coder la fonction `creeCoord` 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).
3. Coder la procédure `afficheCoord` qui prend en paramètre une valeur de type `Coord` et affiche cette valeur sous la forme : (lig, col).
4. Verifier ces fonctions et procédures en utilisant le programme principal suivant :

```
int main(){
  Coord c1 = creeCoord(2,1);
  afficheCoord(c1);
  cout << endl;
  return 0;
}
```

5. Coder les deux fonctions `getX` et `getY`, qui permettent respectivement de récupérer le numéro de ligne et le numéro de colonne d'une coordonnée.
6. Coder la fonction `egalCoord` retournant vrai si deux coordonnées sont égales.
7. Tester cette fonction avec la command `ASSERT` vue en TP dans une fonction `testEgalCoord`.
8. Appeler la fonction de tests ci-dessus dans le main.

2 Compilation séparée

Pour pouvoir le réutiliser, on va placer le code sur les coordonnées dans un module (on dit aussi bibliothèque de fonction) à part. Vous pouvez vous reporter aux explications à la fin du sujet.

9. Créer un fichier d'entête `coord.hpp` avec la définition du type et les entêtes de fonction ;
10. Créer un fichier `testcoord.cpp` avec les différentes fonctions de test et un main qui appelle ces fonctions ;
11. Ne garder que le code des fonctions dans le fichier `coord.cpp` ;
12. Si vous avez configurer votre `Makefile` corectement, vous pouvez compiler le tout avec `make testcoord`
13. Exécuter et vérifier que tout marche.

3 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.

14. Coder le type enuméré `Direction`.
15. Coder la procédure `afficheDirection` qui prend en paramètre une valeur de type `Direction` et affiche cette valeur.
16. Coder les fonctions `aGauche` et `aDroite` qui prennent en paramètre une direction et retournent la direction située juste à sa gauche (respectivement droite).
17. Tester systématiquement ces fonctions, en partant de n'importe quelle direction, avec les scénarios suivants :
    (a) si l'on tourne à gauche puis à droite, on doit être revenu dans la direction initiale ;
    (b) si l'on tourne 8 fois à gauche, on doit être revenu dans la direction initiale ;
    (c) si l'on tourne 8 fois à droite, on doit être revenu dans la direction initiale ;
18. Coder maintenant la fonction `devantCoord` qui retourne la coordonnée devant une coordonnée donnée dans une direction donnée.
19. Dans le fichier de test `testcoord.cpp`, écrire une fonction `testDevantCoord` qui teste quelques coordonnées et appeller cette fonction dans le main de `testcoord.cpp`. Vérifier qu'il n'y a pas d'erreur.
20. 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.

4 Grille

On s'intéresse maintenant au type abstrait `Grille`. On rappelle que la grille est un tableau à deux dimensions, que l'on peut supposer pour l'instant carré, de taille égale à 20. 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.

21. Créer les trois fichier `grille.hpp`, `grille.cpp`, `testgrille.cpp` avec les directives `#include` correctes, et les ajouter convenablement dans le `Makefile` (voir le détail ci-dessous).
22. Coder le type `Case`, indiquant si la case contient une brindille ou un termite (avec le numéro du termite si il y en a un, -1 sinon). Si il n'y a ni termite ni brindille, la case est alors considérée comme vide.
23. Coder ensuite le type abstrait `Grille` (qui est donc un tableau 2D de `Case`). Pour éviter les problèmes de passage de paramètre avec les tableaux, on placera ce tableau dans une structure dont le seul champ sera le tableau (on avait déjà utilisé cette astuce dans la déclaration du type `Polynome` dans l'exercice 1 du TP 8).
24. Coder les procédures et fonctions vues en TD, décrites dans la partie 2.2 du sujet du projet. Il est nécessaire de tester à l'aide de la commande `ASSERT` la faisabilité des procédures et fonctions selon les conditions de l'énoncé. Par exemple, on ne pose une brindille ou un termite dans une case que si cette case est vide.
25. Tester en profondeur chacune des procédures et fonctions écrites. Garder trace de vos tests en les écrivant dans des fonctions de test dans le fichier `testgrille` afin de pouvoir les relancer si vous modifiez votre code.
26. 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.

A Compilation séparée et `Makefile`

Vous avez déjà utilisé compilation séparée et Makefile dans les TP 7 et 8, mais avec des fichiers déjà données dans une archive. Pour que vous puissiez mettre en place cela par vous-mêmes pour le projet, nous vous l'expliquons plus en détail dans cette section. N'hésitez pas à aller revoir les fichiers des TP 7 et 8 (type abstrait polynôme) pour avoir des exemples vous aidant à bien comprendre les principes décrits dans cette section.

L'utilitaire `make` sert à compiler un projet sans avoir à taper les commandes de compilation soi-même. Pour l'utiliser il faut :

— Écrire un fichier `Makefile` qui contient une description de l'architecture du projet (voir ci-dessous) ;
— Lancer la commande de compilation avec `make nomdufichierexecutable` (par exemple `make testcoord` ou bien `make projet`). Si l'on ne précise pas le fichier (on tape juste `make`), c'est le premier fichier décrit dans le `Makefile` qui est produit (ici `projet`).

Les différents fichiers

Avant de décrire la structure du `Makefile`, on rappelle les rôles des différents fichiers :

— `.hpp` : fichier d'entête qui définit les types et donne la liste des entêtes de fonctions. Pour éviter les problèmes en cas d'inclusion multiple, tout le contenu du fichier doit être placé entre les directives `ifndef`, `define` et `endif` de la manière suivante :

```
#ifndef NOMDUFICHIER_HPP
#define NOMDUFICHIER_HPP

// Fichiers systèmes utilisé dans le .hpp
#include<string>

// Fichier définissant les types dont on a besoin
// Ici je suppose que le fichier truc.hpp défini un type Truc
#include "truc.hpp"

// Définition des types
struct Machin {
   std::string nom;
   Truc tr;
};

// Entête des fonctions
/** cree un machin à partir de son nom et de son truc
 @param[in] n : le nom du machin
 @param[in] t : le truc du machin
**/
Machin creeMachin(std::string n, Truc t);

/** retourne le truc d'un machin
 @param[in] a : un machin
**/
Truc getTruc(Machin a);

#endif
```

On rappelle que seuls les fichiers `.hpp` sont inclus par `#include`. On n'inclut jamais un fichier `.cpp`. D'autre part, les fichiers du système sont inclus avec des chevrons et en général sans extension `.hpp` (comme dans `#include <iostream>`). Les fichiers du projet sont inclus par des guillemets (comme dans `#include "coord.hpp"`).

— `.cpp` : fichier qui contient le code source des fonctions. Il doit inclure le fichier `.hpp` correspondant entre guillemets, ainsi que les fichiers d'entête des modules qu'il utilise. Par exemple, au début de `grille.cpp`, qui utilise le module `coord`, on doit trouver

```
1 #include <iostream>
2 #include "coord.hpp"
3 #include "grille.hpp"
```

— `.o` : fichier objet. C'est un fichier qui contient le code compilé d'une bibliothèque de fonctions. Il est obtenu par le compilateur à partir du `.cpp`. Pour obtenir un exécutable, on lie ces fichiers au programme principal.

La structure du `Makefile`

Dans le fichier `Makefile`, Il faut commencer par indiquer le compilateur et les options utilisées. Puis il faut décrire les **dépendances de chaque composant**. Dans notre cas, les composants sont de deux sortes :

— Les exécutables (projet, code de tests). Ils dépendent du fichier `.cpp` correspondant ainsi que des `.o` de tous les composants qu'ils utilisent. Seul le fichier `.cpp` de l'éxécutable principal doit contenir une fonction `main`.
Par exemple, l'executable `testgrille` dépend de `testgrille.cpp` ainsi que de `grille.o` et `coord.o`. Dans le Makefile, on aura donc une ligne `testgrille: testgrille.cpp grille.o coord.o`
De plus, `testgrille.cpp` doit contenir une fonction `main` tandis que `grille.cpp` et `coord.cpp` ne doivent pas en contenir.

— Les bibliothèques (fichier `.o`). Elles dépendent du fichiers `.cpp` correspondant, ainsi que de tous les `.hpp` qui sont inclus. Par exemple, la bibliothèque `grille` dépend de `grille.cpp` qui inclue `grille.hpp` et `coord.hpp`. Dans le Makefile, on aura donc une ligne `grille.o: grille.cpp grille.hpp coord.hpp`

Finalement, on peut donner des commandes annexes, par exemple pour faire le ménage. **Attention :** Quand on donne une commande annexe, on doit aller à la ligne et commencer la ligne contenant la commande par un caractère de **Tabulation** (la touche que l'on utilise jamais assez, à gauche du clavier au dessus des touches de majuscules).

Voici un exemple (incomplet) de `Makefile` :

```
# Fichier makefile pour projet termite
#####################################
##
# Tout ce qui est après un # sur une ligne est en commentaire.
##

# Quelques variables de configuration du compilateur
####################################################
# Le compilateur à utiliser
CXX = g++
# Les options du compilateur
CXXFLAGS = -Wall -std=c++11 -g

# Les programmes principaux
###########################
# On donne le fichier .cpp et la liste des fichiers .o qu'ils utilisent

projet: projet.cpp coord.o grille.o termite.o # autre fichiers utiles
testcoord: testcoord.cpp coord.o
testgrille: testgrille.cpp grille.o coord.o

# Les différents composants
###########################
# on donne le fichier .cpp ainsi que la liste
# des fichiers .hpp dont ils dépendent

coord.o: coord.cpp coord.hpp
grille.o: grille.cpp grille.hpp coord.hpp

# Pour faire le ménage
clean:
	rm -f projet test *.o
# Attention dans la ligne ci-dessus il faut écrire
# un seul caractère de tabulation et pas 8 espaces.
```