ISIMA 3ème année - MODL/C++
TP 1 : Rappels

Préambule

Ce TP vous fera (re)découvrir deux outils de développement logiciel : **CMake** et les **test unitaires**.

CMake (*Cross-platform Make*) est un outil permettant de compiler vos codes sur diverses plateformes. Son rôle est de générer le *makefile* approprié en tenant compte de la configuration particulière de la machine sur laquelle il est exécuté (chemin vers les bibliothèques, options de compilation pour une architecture donnée…). Il fonctionne avec un fichier de configuration qui vous est donné pour ce TP : `CMakeLists.txt`. Nous vous encourageons fortement à le consulter pour comprendre son fonctionnement (la documentation de référence de CMake se trouve à l'adresse suivante : `http://www.cmake.org`).

Pour ce TP, vous n'avez plus qu'à générer le *makefile* de votre projet. Utilisez les commandes suivantes depuis le répertoire où se trouve le fichier `CMakeLists.txt` :

```
cd build
cmake ..
```

À présent vous pouvez taper `make` dans le répertoire `build` pour reconstruire le projet après chaque modification du code source. Vos fichiers sources doivent être placés dans le répertoire `src`. **N'oubliez pas de les déclarer dans le fichier de configuration `CMakeLists.txt`.** Un fichier avec la fonction principale `main` a été préparé pour chaque exercice (cf. `src/main_*.cpp`).

Lors de sa première exécution, `make` est relativement lent. En effet, il compile la **bibliothèque de tests unitaires Catch** (pour plus d'informations : `http://github.com/catchorg/Catch2`). Les tests unitaires sont utilisés pour vérifier qu'un élément logiciel (fonction, méthode, objet...) produit les résultats que l'on attend de lui. On parle de *Test-Driven Development*, dès lors que les tests unitaires sont écrits en amont du code source et guident le développement de l'application. C'est au développeur d'identifier les cas où l'application peut être mise en échec.

Dans notre cas, des tests élémentaires vérifiant le bon fonctionnement des questions de l'exercice 1 vous sont fournis. Vous les trouverez dans le répertoire `test` du projet. La syntaxe des tests est extrêmement simple, il suffit de donner un nom au test et à la batterie qui le contient, avant de faire des assertions pour vérifier le résultat d'un appel de méthode par exemple. **Il faut bien sûr que l'interface du code que vous développez respecte celle utilisée dans les tests**. Par exemple, le code suivant :

```
TEST_CASE ( "TP1_Polaire::Constructeur" ) {
 const double a = 12.0;
 const double d = 24.0;

 Polaire p(a,d);

 REQUIRE ( p.getAngle() == Approx(a) );
 REQUIRE ( p.getDistance() == Approx(d) );
}
```

vérifie que le constructeur de la classe **`Polaire`** remplit correctement les attributs des instances qu'il crée et que les accesseurs `getAngle()` et `getDistance()` renvoient les valeurs attendues.

Exercice 1 - Point

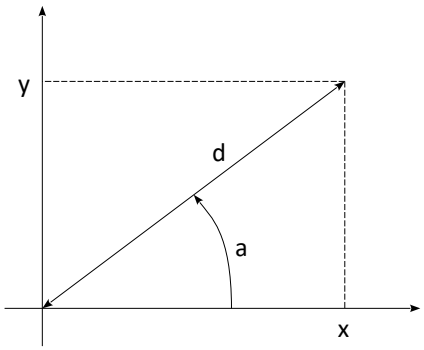
1) Définir une classe **`Point`** proposant une méthode abstraite `afficher()`. Dériver deux classes, **`Cartesien`** et **`Polaire`**, représentant un point sous forme de coordonnées respectivement cartésiennes et polaires (angle en degrés). Offrir un jeu de fonctionnalités minimal à ces classes. Tests 1-11

2) Surcharger l'opérateur de flux de sortie de manière à ce qu'il exploite l'héritage défini précédemment (cf. méthode afficher()). Test 12

3) Proposer des passerelles de conversion entre les deux classes concrètes. Deux approches sont possibles, il vous est demandé de les implémenter :

   (i) Proposer des méthodes virtuelles dans la classe **`Point`** et ses filles pour convertir un point dans chacun des types **`Cartesien`** et **`Polaire`**. Un problème d'interdépendance entre classes vous obligera à faire des déclarations anticipées (*forward declarations*). Tests 13-15

   (ii) Proposer un constructeur dans chacune des deux classes concrètes qui permette la conversion à partir de l'autre classe. Tests 16-17




```
a = std::atan2(y, x) * 180 / M_PI
d = std::hypot(x, y)
x = d * std::cos(a * M_PI / 180)
y = d * std::sin(a * M_PI / 180)
```

https://fr.wikipedia.org/wiki/Coordonn%C3%A9es_polaires

4) Dans le programme principal, stocker des instances de **`Cartesien`** et de **`Polaire`** dans un vecteur STL et afficher son contenu. Essayer `std::vector<Point>` et `std::vector<Point *>`.

5) Aux vues de la question précédente, définir une classe **`Nuage`** utilisant la classe `std::vector` et contenant un ensemble de points (qui peuvent être de types différents). Cette classe devra fournir des itérateurs, à la manière STL (méthodes `begin()` et `end()`, type `iterator`), et une méthode `size()` qui retourne le nombre de points stockés. Réutiliser au maximum les méthodes déjà existantes de la classe `std::vector`. Tests 18-19

6) Proposer une fonction qui calcule le barycentre d'un nuage de points. Tests 20-21

   Proposer ensuite deux versions foncteurs de cette fonction qui renvoient le barycentre sous la forme respectivement d'un cartésien et d'un polaire. Tests 22-23

Exercice 2 - Vecteur *from scratch*

1) Définir une classe **`Vecteur`** conservant des entiers dans un tableau dynamique (lorsque le tableau est plein, sa taille doit être doublée, à l'image du comportement d'un `std::vector`). Proposer les méthodes nécessaires au fonctionnement de cette classe, en particulier celles de la forme normale de Coplien (constructeur par défaut, constructeur par copie, destructeur et opérateur de copie).

2) Ajouter des opérateurs pour l'insertion sur le flux en sortie (`<<`), la concaténation (`+`), l'accès direct (`[]`, accès lecture ou lecture/écriture) et le produit scalaire (`*`).

3) Définir une classe **`Iterateur`** pour parcourir un vecteur avec les mécanismes élémentaires :

   - itérateur au début du vecteur (fourni par le vecteur) : `Iterateur begin()`
   - itérateur à la fin du vecteur (fourni par le vecteur) : `Iterateur end()`
   - passage à l'élément suivant (préfixé) : `Iterateur & operator++()`
   - passage à l'élément suivant (postfixé) : `Iterateur operator++(int)`
   - accès à l'élément pointé : `int operator*()`
   - comparaison entre deux itérateurs : `bool operator==(const Iterateur &)`