TP11 POO - C++

gilles.scarella@univ-cotedazur.fr, simon.girel@univ-cotedazur.fr

1 Exercice sur les fonctions et classes Template

Indication: Pour compiler et exécuter le code de cet exercice, on exécutera dans le terminal

```
g++ main_sort.cpp -o main_sort
./main_sort
```

On se propose d'implémenter un algorithme de tri sur des tableaux de type générique. Pour cela, on va utiliser un des algorithmes de tri les plus simples (loin d'être le plus performant !) appelé "tri à bulle" (déjà cité à la fin du TP8). Le pseudo code de cet algorithme est disponible dans le lien suivant (c'est le 1er algo de la page):
Source Wikipédia : `http://fr.wikipedia.org/wiki/Tri_à_bulles`

1.1 Fonction *MySwap*

- Pour cela, dans un fichier *MySort.hpp*, commencer à écrire une fonction patron *MySwap* sur un type générique T, prenant en paramètre deux références de ce type T, et échangeant leur valeur.
- Tester dans une fonction *main* définie dans le fichier *main_sort.cpp*, sur au moins deux types.

1.2 Fonction *MySort*

- Ecrire une fonction *MySort* implémentant l'algorithme présenté dans le lien Wikipédia. Ce sera une fonction template sur un type générique T, prenant en paramètre un pointeur sur ce type, ainsi que la longueur de ce tableau (un entier).
- Tester cette fonction de tri dans le fichier *main_sort.cpp* avec des tableaux de type divers : un tableau d'entiers, de réels et de chaînes de caractères (*string*) (pour les *string*, on ajoutera #include<string> au début du fichier contenant le *main*).

Indication: Pour comparer deux *string*, les premières lettres de chaque *string* sont utilisées et la comparaison s'appuie sur l'ordre alphabétique. De plus, toute majuscule est considérée plus petite que toute autre minuscule.

2 Héritage et polymorphisme

Indication:

- Dans cet exercice, vous pouvez exécuter ce qui suit dans le terminal, afin de compiler uniquement le code de la classe *Cercle* (où N est la version - même principe pour *Forme*)

  ```
  g++ -c CercleN.cpp
  ```

  Pour compiler la totalité du code et l'exécuter, taper ce qui suit (pour les versions $N = 0$ et $N = 1$):

  ```
  g++ mainN.cpp Point2D.cpp Rot.cpp Mat22.cpp CercleN.cpp FormeN.cpp -o mainN
  ./mainN
  ```

Préliminaire:

- Récupérer sur Moodle, dans le dossier *TP11*, les fichiers *Mat22.cpp*, *Mat22.hh*, *Point2D.cpp*, *Point2D.hh*, *Rot.cpp* et *Rot.hh*.

  Ces fichiers contiennent la définition de la classe *Rot*, modélisant les rotations dans $\mathbb{R}^2$ (avec des matrices 2x2) et qui s'utilise comme suit:

  ```
  Rot r(M_PI/4); // Rotation de pi/4
  Point2D P = {1,0};  // Point dans R^2
  Point2D Q = r(P);   // on obtient Q = {sqrt(2)/2, sqrt(2)/2};
  ```

Le but de cet exercice est de modéliser au moins une forme géométrique en utilisant l'héritage en C++.
Les formes géométriques considérées dans cet exercice seront très simples (penser à un cercle, carré, losange, ...).

2.1 1ère version (V0): héritage standard

2.1.1 La classe *Forme*

Dans le fichier *Forme0.hh*, nous allons déclarer la classe *Forme* de laquelle hériteront les autres formes géométriques. Cette classe modélise une forme géométrique générale.
Dans cette version, nous appliquerons une matrice de rotation aux objets de *Forme* et de sa classe dérivée.

- Dans *Forme0.hh*, déclarer la classe *Forme* telle que
  - La classe *Forme* ne contient qu'une seule donnée membre *_centre* de type *Point2D*, **déclarée *protected***.

- *Forme* possède un constructeur public à un seul paramètre de type *Point2D* (utiliser de préférence une référence constante à un objet *Point2D*).

Forme possède les deux fonctions **membres publiques** suivantes:

- Une fonction *to_str* renvoyant une variable de type *string*, ne prenant rien en paramètre et pouvant s'appliquer à des objets constants
- Une fonction *apply_rot* ne renvoyant rien et prenant une référence constante sur un objet de type *Rot* en paramètre. Comme son but est d'appliquer une rotation à la forme, cette fonction ne sera pas spécifiée *const*.

- Dans le fichier *Forme0.cpp*, écrire les définitions correspondant aux déclarations précédentes pour la classe *Forme*. Plus précisément,
  - Par exemple, la fonction *to_str* de *Forme* renverra la chaîne de caractères suivantes (à adapter selon le centre de la forme; on pourra utiliser la fonction *to_string* de la librairie standard C++ qui permet de convertir une valeur numérique en chaîne de caractères):

    ```
    Forme géométrique de centre (2.5, 1)
    ```

  - la fonction *apply_rot* applique une rotation au centre de la forme géométrique (cf indication précédente sur la classe *Rot*). Le centre de la forme est modifié par la rotation.

2.1.2 La classe *Cercle*

- La classe *Cercle* hérite publiquement de la classe *Forme* et possède un attribut privé supplémentaire de type *double* appelé *_rayon*, correspondant au rayon du cercle
- *Cercle* possède un constructeur public à deux paramètres de type respectif *Point2D* et *double* (le 1er paramètre définit le centre et le second le rayon du cercle)
- On conserve, dans *Cercle*, la version de *apply_rot* héritée de *Forme*.
- Dans *Cercle*, on souhaite redéfinir la fonction membre *to_str* et la spécifier. Par exemple, le résultat de *to_str* pour un objet Cercle sera (à adapter selon le cercle)

  ```
  Cercle de centre (2, 1) et de rayon 1.5
  ```

- Dans le fichier *Cercle0.hh*, déclarer la classe *Cercle* à partir des indications précédentes
- Dans le fichier *Cercle0.cpp*, écrire les définitions du constructeur de *Cercle* et de *to_str*.
  Indication: Utiliser le constructeur de *Forme* dans la 1ère ligne du constructeur de *Cercle*

2.1.3 La fonction *main*

- Dans la fonction *main* du fichier *main0.cpp*, définir deux objets: un objet *Forme* et un objet *Cercle* avec les paramètres que vous voulez. Faites afficher les deux objets en appelant *to_str* sur chacun d'entre eux, puis appliquer à chaque objet la matrice de rotation correspondant à l'objet r de votre choix.

- Il faudra aussi déclarer une variable v comme un tableau de 2 pointeurs sur *Forme*, comme suit:

  ```
  Forme* v[2];
  ```

  Le premier élément de v pointera vers l'objet de type *Forme* et le second élément de v pointera vers l'objet de type *Cercle*. Faire afficher les éléments de v en utilisant la fonction *to_str*.
  Attention : Les éléments de v sont des pointeurs! Pour appeler les fonctions membres des objets pointés, il faudra utiliser l'opérateur $\rightarrow$

- Exécuter le code. Que remarquez-vous sur le résultat de l'appel à *to_str* sur les variables de v ?

2.2 2ème version (V1): avec une fonction virtuelle

Cette deuxième version est très proche de la version V0 et le code demandé ressemble beaucoup à la version précédente.

2.2.1 La classe *Forme*

Les spécifications de la classe *Forme* sont identiques à la version précédente V0 à l'exception que, désormais, la fonction *to_str* doit être déclarée *virtual*.

- Déclarer la classe *Forme* dans le fichier *Forme1.hh* en modifiant la déclaration de *to_str* (c'est la seule modification)

- Définir la classe *Forme* dans le fichier *Forme1.cpp* (il faut inclure *Forme1.hh* - c'est la seule modification)

2.2.2 La classe *Cercle*

Les spécifications de la classe *Cercle* sont identiques à la version précédente V0 mais on utilise désormais la classe *Forme* déclarée dans *Forme1.hh*.

- Déclarer la classe *Cercle* dans le fichier *Cercle1.hh* (il faut inclure *Forme1.hh* - c'est la seule modification)

- Définir la classe *Cercle* dans le fichier *Cercle1.cpp* (il faut inclure *Cercle1.hh* - c'est la seule modification)

2.2.3 La fonction *main*

- Dans le fichier *main1.cpp*, vous devez utiliser exactement les mêmes variables que dans le *main* de la version précédente à l'exception qu'il faut inclure *Cercle1.hh* (qui contient *Forme1.hh*)
- Exécuter le code. Quelle différence voyez-vous dans le résultat de l'exécution par rapport à la version précédente?

2.3 3ème version (V2): *Forme* devient une classe abstraite

2.3.1 La classe *Forme*

Désormais, *Forme* devient une classe abstraite, il n'existera donc pas d'objet de la classe *Forme* (mais seulement des pointeurs sur *Forme*). Pour résumer:

- ◦ On conserve l'attribut *protected* *_centre* de *Forme*
- ◦ *Forme* ne possède plus de constructeur
- ◦ *to_str* et *apply_rot* deviennent virtuelles pures dans *Forme*
- Dans le fichier *Forme2.hh*, modifier la déclaration de la classe *Forme*, en suivant ce qui vient d'être indiqué.
- Il n'y a plus besoin de définir les fonctions membres de *Forme* qui est une classe abstraite.

2.3.2 La classe *Cercle*

Dans cette version, la classe *Cercle* hérite publiquement de la classe *Forme* (celle déclarée dans *Forme2.hh*).
Les spécifications de la classe *Cercle* sont assez semblables aux deux versions précédentes, à l'exception de ce qui suit:

- ◦ Il faut inclure désormais *Forme2.hh* dans *Cercle2.hh*
- ◦ Il faut déclarer *apply_rot* pour *Cercle*
- Dans le fichier *Cercle2.hh*, déclarer la classe *Cercle*
- Dans le fichier *Cercle2.hh*, définir le constructeur de *Cercle* ainsi que les deux fonctions membres *to_str* et *apply_rot* pour *Cercle*

2.3.3 La fonction *main*

- On ne peut plus instancier désormais de variable *Forme*
- Vous pouvez utiliser la même variable de type *Cercle* que dans les *main* des exemples précédents, à l'exception qu'il faut inclure *Cercle2.hh* (qui contient donc *Forme2.hh*)

- Dans la fonction *main* du fichier *main.cpp*, déclarer une variable v comme un pointeur sur *Forme* et qui pointe vers l'objet *Cercle*
- Exécuter le code, vous devez obtenir un affichage semblable à la version précédente pour l'objet *Cercle*

2.3.4 [FACULTATIF] La classe *Carre*

- Créer une classe *Carre*, héritant de *Forme*, possédant l'attribut privé supplémentaire *_pts*, tableau de quatre *Point2D*, correspondant aux sommets du carré.
  apply_rot devra gérer la rotation du centre du carré ainsi que celle de ses sommets.
  to_str devra produire un affichage représentatif d'un objet *Carre*.
- Dans le *main*, v redevient un tableau de 2 pointeurs sur *Forme* et son deuxième élément pointera alors vers un objet de type *Carre*. Tester le code.