C++ — TP 1

1 Préambule

Dans ce premier TP nous allons aborder :
— Les namespaces
— Le principe d'ADL (*Argument-Dependent name Lookup*)
— Le polymorphisme dynamique et statique

2 Premiers pas

L'écriture d'un code, quel qu'il soit, doit être pensée pour sa réutilisation et sa maintenance. Un code existant doit être facilement extensible et l'ajout de nouvelles fonctionnalités ne doit pas induire une modification du code existant. Pour ce faire, le polymorphisme nous permet de réutiliser du code existant sur différents types. Au sein de C++, le mécanisme d'héritage nous permet de réaliser du polymorphisme de classe.
Pour traiter les problèmes listés dans le préambule, nous allons travailler avec les formes géométriques, nous les nommerons *shapes*. Grâce au propriétés mathématiques des *shapes* nous allons illustrer et démontrer comment réaliser de l'héritage proprement.

3 Implémentation

3.1 La classe de base

Notre classe de base représentera le *shape* rectangle, elle sera nommée `rectangle`. Elle respectera la forme canonique de Coplien et proposera des accesseurs et des mutateurs. De plus, elle devra proposer une fonction membre de calcul de l'aire du rectangle.

— **Comment doit ont déclarer les fonctions membres pour que la classe de base soit dérivable ?**

— **Quelles fonctions membres de notre classe doivent respecter cette déclaration ?**
— **Implémentez cette classe !**

3.2 La classe dérivée : Polymorphisme dynamique

La classe dérivée de `rectangle` sera la classe `square` (carré) qui héritera dynamiquement de la classe `rectangle`.

— **Quelles fonctions membres doivent être redifinies dans notre nouvelle classe ?**
— **Quelles sont les limitations d'un tel héritage ?**
— **Implémentez cette classe !**

3.3 Asserts et exceptions

Lors de la construction de la classe dérivée `square` et durant l'appel de ses fonctions membres une précondition et une postcondition doit être toujours vrai.

— **Quelles sont ces pré/postconditions ?**

Pour gérer cette programmation par contrat, des asserts sont mises en place pour les préconditions et des exceptions sont envoyées si la/les postconditions ne sont pas vérifiées.

— **Durcissez vos classes en respectant ce principe.**

3.4 Namespace, ADL et Swap

Les *namespaces* sont des espaces de nommage dans lesquels classes et fonctions vivent. En effet, si deux fonctions sont déclarées avec le même nom, un conflit apparaît. Les *namespaces* permettent de qualifier ces fonctions (ou classes) et le compilateur sera capable de distinguer les deux déclarations. Lors de l'appel de cette fonction non qualifiée, l'*Argument-Dependent name Lookup* (ADL) va se charger de trouver le bon appel de fonction parmis les namespaces qualifiants les arguments de la fonction. Attention, l'ADL ne survient que lorsque la phase classique de *lookup* échoue.

— **Implémentez la surcharge de l'opérateur `<<` pour afficher un rectangle ou un carré dans la console. Illustrer l'ADL réalisé lors de l'appel de cet opérateur.**
— **Implémentez une fonction libre de `swap` sur les rectangles/carrés en utilisant Boost SWAP.**
— **Expliquez comment l'ADL est utilisé au sein de Boost SWAP.**

3.5 LSP : Liskov Substitution Principle

Le LSP est un principe de programmation, son énoncé est le suivant :

"Si un objet x de type `T` est attendu alors on doit pouvoir passer tout objet y de type `U`, `U` dérivant de `T`."

Dans notre exemple, le LSP s'exprime de la manière suivante : si une fonction prend en argument un `rectangle` on doit pouvoir passer un `carré` à cette même fonction. Pour vérifier le LSP, nous allons implémenter la fonction `test_area`. Cette fonction prendra un `rectangle` en argument, génèrera deux nombres entiers aléatoires et appelera la fonction membre `area` sur son argument. Dans le corps de la fonction, il faudra vérifier que l'aire renvoyée par la fonction membre `area` est bien égale à l'aire attendue. Si ce n'est pas le cas, une exception sera envoyée.

— **Codez cette fonction.**
— **Appelez cette fonction avec un `carré` en paramètre. Expliquez pourquoi le LSP n'est pas respecté.**