Enseignant : J.Falcou (joel.falcou@lri.fr)
Resp. TP : F.Laguzet (florence.laguzet@lri.fr)

TP1 : La méthode LISKOV et le Design Pattern Fonctor

Première partie
Utilisation de la méthode LISKOV

1 Préambule

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

Dans un premier temps vous allez procéder à l'**installation des bibliothèques BOOST** disponibles à l'adresse suivante : `http://www.boost.org/users/download/`.

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 précondition 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 classe) 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 qualifiant les arguments de la fonction. Attention, l'ADL ne survient que lorque 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.**

4 Bonus

4.1 Héritage

– **Proposez un autre arbre d'héritage permettant de respecter le LSP.**

4.2 Polymorphisme statique

Le polymorphisme dynamique trouve sa limitation lorsque le code à l'intérieur de ces fonctions est court et/ou lorsque ces fonctions sont appelées souvents. Lors d'un appel de fonction classique, le compilateur résoud cet appel et génére un saut directement à l'adresse de cette fonction. Dans le cas des fonctions virtuelles, le compilateur ne peut pas savoir quel saut réaliser pour tomber dans le bon code de la fonction car le polymorphisme dynamique induit une resolution de l'appel de la fonction au runtime (fonction virtuelle). Cet résolution dynamique utilise une table des fonctions virtuelles. A chaque appel d'une fonction

virtuelle, il faut alors aller chercher l'adresse de la fonction correspondante dans la table ce qui a un coût. Pour éviter ce surcoût, le polymorphisme statique permet de réaliser cette résolution à la compilation.

– **Réimplémentez le nouvel arbre d'héritage en utilisant le polymorphisme statique.**

Deuxième partie
Les *Fonctor* et le calcul d'intégrales

5 Implémentation du calcul de l'intégrale

Nous allons implémenter le calcul de l'intégrale pour une fonction f choisie. Pour cela, nous allons avoir plusieurs manières concernant l'implémentation du calcul de la fonction f : en tant que pointeur de fonction, d'objet contenant une méthode et enfin à l'aide d'un *Fonctor*.
Vous allez donc être amenés à coder 3 fonctions différentes :

compute_ptr Fonction prenant en argument un pointeur de fonction et retournant un double

compute_class Fonction prenant en argument un objet et retournant un double

compute_fonc Fonction prenant en argument un *Fonctor* et retournant un double

Vos fonctions auront donc comme prototype :
double `compute_X( < fonction` f `>, double X, double` δ_x`)`.
L'intervalle sur laquelle nous allons calculer l'intégrale sera $[X - \delta_x, X + \delta_x]$.
La fonction f, quant à elle, est définie comme prenant en paramètre un double et renvoyant un double.
Que remarquez-vous concernant les 3 manières de passer f en paramètre ?

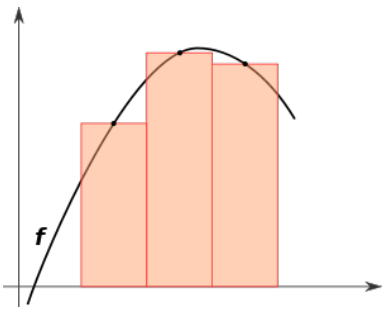
6 Temps d'exécution

Ensuite, vous allez calculer le temps d'exécution pour chacune des trois méthodes (pointeur, objet et *Fonctor*). Pour cela, vous utiliserez la fonction `gettimeofday` définie dans `#include <sys/time.h>`.
Pour son utilisation, se reporter au manuel (`man gettimeofday`).
Que remarquez-vous concernant les temps d'exécution ?

7 Rappel : Formule de l'intégration numérique

Vous trouverez à l'adresse suivante un rappel concernant le calcul d'intégrales par la méthode des rectangles : *http ://homeomath.imingo.net/methrect.htm.*
Vous pouvez très bien utiliser une autre méthode (trapèze, Simpson, ...).

Soit f la fonction à intégrer et a b les bornes. On a alors $I(f) = (b-a)f(\xi)$ avec f la fonction pour laquelle nous voulons calculer l'intégrale.



Source : Wikipedia