TP : Quelques Patterns Structurels En C++

Récupérer le code à compléter sur https://www.lri.fr/~bagneres/index.php?section=teaching&page=2016_App4_poo.

1 Hiérarchie De Classes & Relation Entre Classes

On dispose d'un parseur pour les expressions écrites en notation polonaise inversée. Pour parser une expression, on créé une factory depuis la fonction `make_factory` et on passe une chaîne de caractères à sa fonction membre `make` (voir l'exemple dans `tests/make_factory.cpp`).

1.1 Dessiner le diagramme UML des classes `expression_t`, `constant_t`, `variable_t` et `operator_t` comme si la classe `expression_t` était la classe mère (au lieu de `expression_t_`).

1.2 Pourquoi ces classes correspondent au pattern *composite* ? Décrire les différentes parties et les "connexions" entre ces classes.

1.3 À quel pattern structurel correspond la classe `expression_t` ? Expliquer pourquoi.

1.4 Dessiner le diagramme UML exact des classes `expression_t_`, `constant_t`, `variable_t`, `operator_t` et `expression_t_`.

1.5 Commenter ce diagramme en le comparant au premier.

2 Évaluation D'Expressions

On cherche à évaluer les variables d'un ensemble d'expressions comme dans le test `tests/eval.cpp`.

2.1 Pourquoi l'appel de la fonction membre `expression_t::eval` ne répond pas au problème ?

On propose d'écrire la fonction libre `eval` qui :
- prend en paramètre un ensemble d'expressions à parser (dans un `std::vector<std::string>`)
- retourne les valeurs des variables dans une `std::map<std::string, double>`
  http://en.cppreference.com/w/cpp/container/map
  La clé (`key_type`) correspond au nom de variable et la valeur (`mapped_type`) à sa valeur

Cette fonction :
- recupère un `expression_t` depuis un `std::string`
- propage les valeurs actuelles des variables grâce à la fonction libre `propagate`
- teste si l'expression est un `operator_t` grâce à la fonction `is_operator`
- récupère une référence sur `operator_t` (`auto & op = expression_cast<operator_t>(e);`)
- teste si l'expression est une affectation dans une variable
- évalue l'expression de droite et enregistre la valeur pour cette variable

La fonction libre `propagate` prend en paramètre les valeurs des variables actuelles dans un `std::map<std::string, double>` et l'`expression_t` à modifier. Cette fonction va parcourir l'`expression_t` et mettre à jour les valeurs des variables de l'`expression_t` en fonction de la `std::map`.

2.2 Dans le fichier `tests/eval.hpp`, écrire le corps des fonctions `eval` et `propagate`.

2.3 Cette solution est mauvaise. Pourquoi ?

2.4 Quel mécanisme fourni par les langages orientés objet doit-on utiliser pour régler ce problème ? (Ne pas coder la solution, on utilisera le pattern visiteur dans un prochain TP.)

3 Proxy

On cherche à écrire une classe se comportant comme un tableau (`std::vector<T>`) mais qui correspond à une sous partie linéaire et contigüe d'un autre tableau.

Comme cette classe se comporte comme un *container* classique. Il est donc possible :
- de parcourir ce container comme n'importe quel tableau
- d'appliquer les fonctions classiques comme le tri (`std::sort`), la somme (`std::accumulate`), la transformation (`std::transform`)

Par exemple, voici le code C++ de `tests/vector_view.cpp` utilisant la classe `vector_view<T>` :

```cpp
// Vector
std::vector<int> vector(10);

// View

auto view = make_view(vector, 2, 7);

for (size_t i = 0; i < view.size(); ++i)
{
    view[i] = int(view.size() - i);
}

std::cout << "View         = " << view << std::endl;   //       { 5, 4, 3, 2, 1 }
std::cout << "Vector       = " << vector << std::endl; // { 0, 0, 5, 4, 3, 2, 1, 0, 0, 0 }
std::cout << std::endl;

// Increment

for (auto & e : view)
{
    ++e;
}

std::cout << "View         = " << view << std::endl;   //       { 6, 5, 4, 3, 2 }
std::cout << "Vector       = " << vector << std::endl; // { 0, 0, 6, 5, 4, 3, 2, 0, 0, 0 }
std::cout << std::endl;

// Sort

std::sort(view.begin(), view.end());

std::cout << "View         = " << view << std::endl;   //       { 2, 3, 4, 5, 6 }
std::cout << "Vector       = " << vector << std::endl; // { 0, 0, 2, 3, 4, 5, 6, 0, 0, 0 }
std::cout << std::endl;

// View of view

auto view_of_view = make_view(view, 1, 3);

std::transform(view_of_view.begin(), view_of_view.end(), view_of_view.begin(), [](int const) {
    return 7; });

std::cout << "View of view = " << view_of_view << std::endl; //         { 7, 7 }
std::cout << "View         = " << view << std::endl;         //       { 2, 7, 7, 5, 6 }
std::cout << "Vector       = " << vector << std::endl;       // { 0, 0, 2, 7, 7, 5, 6, 0, 0, 0 }
std::cout << std::endl;
```

3.1 Dans le langage portable "de votre choix" (contraites en dessous), écrire le code équivalent à `tests/vector_view.cpp`

3.2 Commenter votre solution en détaillant : son fonctionnement (création et utilisation de la vue) ; et les concepts utilisés permettant de s'interfacer avec le langage et sa bibliothèque standard.

Le langage doit être portable et doit fonctionner sous Debian GNU/Linux 8.4 (Jessie) (actuelle *stable*) en ligne de commande. Fournir les commandes de compilation (en activant tous les warnings) et d'exécution dans un fichier `README.txt`.

Si vous choisissez C++11 ou C++14 =) aidez-vous de http://en.cppreference.com/w/cpp/container/vector (et du chargé de TP) pour implémenter les fonctions membres dans `tests/vector_view.hpp`.