TP : Factory en C++

Ce TP propose une solution pour parser des expression en notation polonaise inversée (et simplifiée) https://fr.wikipedia.org/wiki/Notation_polonaise_inverse.

Récupérer le code à compléter sur https://www.lri.fr/~bagneres/index.php?section=teaching&page=2016_App4_prog_parallele.

1 Généralités

En C++, le typage est statique (connue à la compilation). Les objets issues de classes se trouvant dans une hiérarchie comportant de l'héritage peuvent également avoir un type dynamique (connue à l'exécution). Le type dynamique fonctionne uniquement via les pointeurs (« intelligents » ou non) ou les références.

1.1 Quels sont les intérêts du pointeur intelligent `std::unique_ptr<T>` ?

Un `std::unique_ptr<T>`, comme les objets respectant la *sémantique d'entité*[1], ne doit pas être copié mais peut être déplacé.

Ces objets doivent être passés aux fonctions par *r-value reference* (*r* pour *right*) grâce à cette signature :
`void f(type_t && object);`

Pour appeler la fonction `f`, il faut :
- soit lui passer un temporaire (une r-value), par exemple :
  `f(type_t());`
- soit forcer le déplacement de la *l-value* (*l* pour *left*) (souvent une variable), par exemple :
  `f(std::move(a_left_value));`[2]

1.2 Pourquoi est-il dangereux d'appeler `std::move` sur une l-value ? Quels précautions doit-on prendre par la suite ?

2 `expression_t_` & `expression_t`

On propose d'écrire différents types d'expressions : les constantes, les variables et les opérateurs.

Les constantes seront gérées par la classe `constant_t` qui contient un entier `int value` comme donnée membre.
Les variables seront gérées par la classe `variable_t` qui a les données membres suivantes : `std::string name` et `int value`.
La classe `operator_t` a trois données membres : `char symbol`, `expression_t a` et `expression_t b`. Le symbole peut être `+`, `-`, `*` ou `/`. `a` et `b` correspondent respectivement au fils gauche et au fils droit.

Ces trois classes héritent de `expression_t_` qui a les fonctions membres `virtual pure` suivantes :
- `virtual void display(std::ostream & out = std::cout) const = 0;`
  qui affiche l'expression sous une forme facilement lisible par un humain
- `virtual double eval() const = 0;`
  qui donne le résultat du calcul

Afin d'éviter la manipulation un `std::unique_ptr<expression_t_>`, la classe `expression_t` l'encapsule afin que la résolution des appels dynamiques soit transparente et que la différence entre le type statique et dynamique deviennent un détail d'implémentation.

2.1 Pourquoi doit-on manipuler un `std::unique_ptr<expression_t_>` (potentiellement géré par `expression_t`) et pas un `expression_t_` ?

2.2 Compléter le code dans `tests/expression.hpp` afin que le test `tests/expression.cpp` s'exécute correctement.

1. http://cpp.developpez.com/faq/cpp/?page=Les-classes-en-Cplusplus#Quand-est-ce-qu-une-classe-a-une-semantique-d-entite
2. http://en.cppreference.com/w/cpp/utility/move

3 Parser Une Expression

Pour simplifier le parseur, on décide de gérer uniquement les cas où la pile des expressions précédentes est de 2. Cela permet de gérer uniquement deux variables temporaires au lieu d'une `std::stack<expression_t>`.

3.1 Implémenter le constructeur de la classe `expression_t` permettant de construire une expression depuis la chaîne de caractère (`std::string`) passée en argument.

3.2 Pourquoi cette solution est mauvaise ? Quels sont les défauts ?

4 Factory

4.1 Quels problèmes sont résolus par la création d'une factory ?

5 Factory (Version Objet)

Créer une factory `factory_vo_t` avec les fonctions membres suivantes :
- `add` qui prend en paramètre un objet avec deux fonctions membres :
  - `is_convertible` qui prend une chaîne de caractère et retournant le booléen `true` si la chaîne de caractère peut être convertie dans le type retourné par `make`
  - `make` qui prend un paramètre une chaîne de caractère et deux `expression_t` afin de créer le bon `expression_t` depuis cette chaîne de caractères
- `make` qui prend en paramètre une chaîne de caractère et retourne le bon `expression_t`

En se basant sur le test `test/factory.cpp`, créer son propre test afin de tester sa factory.

6 *Factory* (Version C++)

Le fonctionnement de cette factory est similaire à la précédente mais permet de régler les deux "défauts" suivants :
- les objets nécessaires à `factory_vo_t::add` doivent être écrits dans une hiérarchie de classe et ne font qu'encapsuler deux fonctions (*overkill* (?))
- la fonction membre `make` des objets nécessaires à `factory_vo_t::add` attend une signature particulière, si cette signature change, il faut la changer dans l'ensemble des classes filles

La fonction membre `add` de la nouvelle factory `factory_t` demande deux paramètres :
- une fonction (ou un objet fonction ou une lambda fonction) qui prend une chaîne de caractère et retourne `true` si la chaîne de caractère peut être convertie dans le type retourné par la deuxième fonction
- une fonction (ou un objet fonction ou une lambda fonction) qui prend une chaîne de caractère et retourne le bon `expression_t`

On ajoute une deuxième fonction membre `add` qui demande deux paramètres :
- une fonction (ou un objet fonction ou une lambda fonction) qui prend une chaîne de caractère et retourne `true` si la chaîne de caractère peut être convertie dans le type retourné par la deuxième fonction
- une fonction (ou un objet fonction ou une lambda fonction) qui prend une chaîne de caractère et deux `expression_t` et retourne le bon `expression_t`

C'est la factory qui se charge de généraliser la différence entre les paramètres des fonctions membres `add`.
En pratique il suffit de stocker un objet fonction (avec `std::function`) qui prend une chaîne de caractère et les deux `expression_t`. On fait la conversion de la première version vers la deuxième grâce à une lambda fonction.
La fonction membre `make` est identique à celle de `factory_vo_t`, elle prend en paramètre une chaîne de caractère et retourne le bon `expression_t`.

6.1 Pourquoi le deuxième problème de `factory_vo_t` énoncé est résolu ?

7 `make_factory`

Afin de cacher le code nécessaire à la création de la `factory_vo_t`, créer une fonction libre `make_factory_vo_t` qui s'en charge.

En se basant sur le test `test/make_factory.cpp`, créer son propre test afin de tester `make_factory_vo_t`.

8 Améliorations Possibles

— Ajout de l'opérateur =
— Ajout des opérateurs unaires
— Lecture des expressions depuis un fichier
— Évaluation de plusieurs expressions nécessitant l'affectation des variables
— Rendre générique la factory et appliquer une *abstract factory*