TP : Patterns Comportementaux *Visitor* & *Observer*

Généralités

Le code source à rendre peut être écrit dans le langage portable "de votre choix" (contraites en dessous). La solution doit répondre à l'énoncé.

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`.

1 *Composite* & *Visitor*

En partant du TP précédent (ou équivalent), on cherche à remplacer les fonctions libres suivantes par des *visitors* :
- `operator <<` (ou de la fonction membre `display`) permettant l'affichage d'une `expression_t`
- `eval` permettant l'évaluation d'un ensemble d'expressions

1.1 Expliquer le fonctionnement de l'`operator <<` (ou la fonction membre `display`) permettant l'affichage d'une `expression_t`.

1.2 Écrire le *visitor* `display_t` qui permet d'afficher une `expression_t` (sans utiliser la fonction membre `display`).
Indiquer les fichiers créés et modifiés.

Le code suivant (ou équivalent) doit fonctionner :

```
std::vector<std::string> const lines =                                  1
{                                                                       2
    "a 5 =",                                                            3
    "b 2 =",                                                            4
    "c a b + =",                                                        5
    "r c a - 40 + ="                                                    6
};                                                                      7
                                                                        8
auto f = make_factory();                                                9
                                        // Affichage :                  10
display_t display;                                                      11
                                        // (a = 5)                      12
for (auto const & line : lines)         // (a = 5)                      13
{                                                                       14
    auto e = f.make(line);              // (b = 2)                      15
                                        // (b = 2)                      16
    std::cout << e << std::endl;                                        17
    e.accept(display);                  // (c = (a + b))                18
    std::cout << std::endl;             // (c = (a + b))                19
                                                                        20
    std::cout << std::endl;             // (r = ((c - a) + 40))         21
}                                       // (r = ((c - a) + 40))         22
```

Note : en C++, il faut séparer les déclarations et les définitions car la déclaration anticipée ne suffit pas dans notre cas. Utiliser `override`. Attention aux `const` si vous faites du code propre.

1.3 Expliquer votre code et les différentes étapes entre `e.accept(display)` et l'affichage.

1.4 Comparer les deux solutions permettant l'affichage (avantages, inconvénients, limitations, modifications, ajouts ...).

1.5 Écrire le *visitor* `eval_t` qui permet d'évaluer un ensemble d'expressions (vous pouvez utilisez la fonction membre `eval`). Indiquer les fichiers créés et modifiés.

Le code suivant (ou équivalent) doit fonctionner :

```
std::vector<std::string> const lines =
{
    "a 5 =",
    "b 2 =",
    "c a b + =",
    "r c a - 40 + ="
};

auto f = make_factory();

std::vector<expression_t> expressions;

for (auto const & line : lines)
{
    expressions.push_back(f.make(line));
}

eval_t eval(expressions);

std::cout << "Variables = \n" << eval.vars() << std::endl; // { a: 5, b: 2, c: 7, r: 42 }
```

1.6 Expliquer votre code et les différentes étapes entre `e.accept(display)` et l'affichage.

1.7 Comparer les deux solutions permettant l'évaluation (avantages, inconvénients, limitations, modifications, ajouts ...).

1.8 Pour `display_t` et `eval_t`, indiquer si le visiteur a besoin de modifier le visité et si le visité a besoin de modifier le visiteur.

2 *Observer*

On cherche à écrire deux pointeurs :
- le premier, `observable_ptr<T>`, est responsable de la mémoire et informe le deuxième si la donnée qu'il tient change
- le deuxième, `observer_ptr<T>`, pointe sur le premier est accepte les notifications du premier

2.1 Expliquer les notions de pointeurs (ou équivalent), références (ou équivalent) et durée de vie des objets (ou équivalent).

`observable_ptr<T>`

`observable_ptr<T>` peut être construit à partir d'un pointeur à responsabiblité unique (ou équivalent) (`std::unique_ptr<T> &&` en C++).

On peut récupérer un pointeur sur les données avec la fonction membre `get()` et une référence sur les données avec l'opérateur `operator *` (ou équivalent).

On peut changer les données grâce à l'opérateur `operator =` (ou équivalent) (qui prend un `std::unique_ptr<T> &&` en C++).

`observable_ptr<T>` gère aussi plusieurs `observer_ptr<T>`
(dans un `std::vector<std::reference_wrapper<observer_ptr<T>>>` en C++).
Il possède les fonctions membres `add_observer` et `remove_observer` ainsi que `notify` pour mettre à jour l'ensemble des *observers*.

`observer_ptr<T>`

`observer_ptr<T>` peut être construit à partir d'une référence sur un `observable_ptr<T>`.

`observer_ptr<T>` peut changer d'`observable_ptr<T>` avec son opérateur `operator =` (ou équivalent).

Lorsque un `observer_ptr<T>` est détruit avec le destructeur (ou une fonction membre équivalente), il est enlevé des *observers* d'`observable_ptr<T>`.

Comme `observable_ptr<T>`, `observer_ptr<T>` possède la fonction membre `get` et l'opérateur `operator *` (ou équivalent).

`observable_ptr<T>` possède une fonction membre `notify` permettant d'être avertie d'un changement de la donnée tenue par `observable_ptr<T>` (mais pas de la valeur de la donnée).

2.2 Écrire le code correspondant à `observable_ptr<T>` et à `observer_ptr<T>`.

Le code suivant (ou équivalent) doit fonctionner :

```
// observable_ptr<int>
{
    observable_ptr<int> p0 = std::make_unique<int>(7);
    observer_ptr<int> v0(p0); // add_observer
    observer_ptr<int> v1(p0); // add_observer

    std::cout << "p0 = " << p0 << std::endl; // p0 = 7
    std::cout << "v0 = " << v0 << std::endl; // v0 = 7
    std::cout << "v1 = " << v1 << std::endl; // v1 = 7

    *v1 = 42;

    std::cout << "p0 = " << p0 << std::endl; // p0 = 42
    std::cout << "v0 = " << v0 << std::endl; // v0 = 42
    std::cout << "v1 = " << v1 << std::endl; // v1 = 42

```

```
    p0 = nullptr; // observer_ptr<T>::update(): p changed, new value = (null)
                  // observer_ptr<T>::update(): p changed, new value = (null)
} // remove_observer
  // remove_observer
```

```
// observable_ptr<std::string>
{
    observable_ptr<std::string> p0 = std::make_unique<std::string>("An observable_ptr of string");
    observable_ptr<std::string> p1 = std::make_unique<std::string>("A second one");
    observer_ptr<std::string> v0(p0); // add_observer
    observer_ptr<std::string> v1(p0); // add_observer

    std::cout << "p0 = " << p0 << std::endl; // p0 = An observable_ptr of string
    std::cout << "p1 = " << p1 << std::endl; // p1 = A second one
    std::cout << "v0 = " << v0 << std::endl; // v0 = An observable_ptr of string
    std::cout << "v1 = " << v1 << std::endl; // v0 = An observable_ptr of string

    v0 = p1; // remove_observer
             // add_observer

    std::cout << "v0 = " << v0 << std::endl; // v0 = A second one
    std::cout << "v1 = " << v1 << std::endl; // v1 = An observable_ptr of string

    p0.release(); // observer_ptr<T>::update(): p changed, new value = (null)
} // remove_observer
  // remove_observer
```

2.3 Donner un intérêt d'utiliser `observable_ptr<T>` et `observer_ptr<T>` avec un exemple.