TP 4 : Tag Dispatching & SFINAE

Prénom : _____ **Nom :** _____

http://www.boost.org/community/generic_programming.html (C++98)
http://en.cppreference.com/w/cpp/language/sfinae

1 `local::clone()`

En repartant du code du premier TP, on cherche à écrire une fonction générique `local::clone()` qui appelle la fonction membre `.clone()` si elle existe sinon, elle utilisera l'opérateur d'affectation `operator =`.

Dans `tests/copy_ptr.cpp`, désactiver en commentant les tests avec la classe mère et la classe fille.
Ajouter les tests suivants :

```
// Test 0
local::copy_ptr<mother_t> p = daughter_t();
p->fct();

// Test 1
local::copy_ptr<mother_t> p_tmp_mother = daughter_t();
p = p_tmp_mother;
p->fct();
```

1.1 Pour les deux tests, Dans le code `p->fct();`, quelle est la fonction appelée (classe mère ou fille) ? Pourquoi ? Comment corriger ce "problème" ?

Ajouter dans la classe mère et la classe fille la fonction membre `.clone()`. Les signatures de `.clone()` pour la classe mère et la classe fille sont les suivantes :
`virtual std::unique_ptr<mother_t> clone() const`
`virtual std::unique_ptr<mother_t> clone() const override`

1.2 Pourquoi `.clone()` renvoie un `std::unique_ptr<mother_t>` ?

Intégrer `sfinae.hpp` et `is_cloneable.hpp` dans votre projet et créer le fichier `clone.hpp` dans le dossier `include/local`.

1.3 Écrire la fonction `local::clone()` en utilisant un tag dispatching avec `local::is_cloneable`. Quel est son type de retour ?

Remplacer tous les constructeurs de `copy_ptr` par ces constructeurs :

```
// Constructors from nullptr
constexpr copy_ptr() : std::unique_ptr<T>(nullptr) { }
constexpr copy_ptr(std::nullptr_t const) : std::unique_ptr<T>(nullptr) { }
// Constructors from T
template <class U>
copy_ptr(U const & u); // TODO
template <class U>
// Copy (with conversion) constructors
copy_ptr(copy_ptr<U> const & p); // TODO
copy_ptr(copy_ptr<T> const & p); // TODO
// Move constructor
copy_ptr(copy_ptr<T> &&) = default;
template <class U>
// Constructors from std::unique_ptr<T>
copy_ptr(std::unique_ptr<U> && p) : std::unique_ptr<T>(p) { }
copy_ptr(std::unique_ptr<T> && p) : std::unique_ptr<T>(p) { }
```

Remarque : si l'opérateur d'affectation utilisait l'idiome « copy and swap », son implémentation restera correcte.

1.4 Écrire (après avoir lu toute la question) la définition des trois constructeurs non implémentés. Pourquoi l'utilisation de `local::clone` n'est pas commode/adaptée ? Utiliser la fonction (à écrire) `local::clone_to_unique_ptr`.

Ajouter le test suivant :

```
local::copy_ptr<daughter_t> p_tmp_daughter = daughter_t();
p = p_tmp_daughter;
```

1.5 Pourquoi ce test ne compile pas ? Comment résoudre le problème ?

2 Périmètre d'une forme géométrique

On cherche à calculer le périmètre d'un carré ou d'un rectangle en utilisant le tag dispatching.

On dispose des types tags suivants :

```
class square_tag { };
class rectangle_tag { };
```

Écrire les classes `template square_t` et `rectangle_t`.
`square_t<T>` contient un type membre `public square_tag`, une donnée membre `public side`, le constructeur par défaut et celui qui prend un `T const &`.
`rectangle_t<T>` contient un type membre `public rectangle_tag`, deux données membres `public side_a` et `side_b`, le constructeur par défaut et celui qui prend deux `T const &`.
Pour les deux classes, surcharger l'opérateur `«` avec un `std::ostream`.

2.1 Pourquoi les données membres `side`, `side_a`, `side_b` peuvent être `public` ?

2.2 Quel est le type de retour des fonctions `perimeter` ?

2.3 Ici, par quel autre mécanisme plus simple le tag dispatching peut être remplacé ?

3 Une fonction `local::random(a, b)` générique

On cherche à compléter la fonction `local::random(a, b)` pour la rendre générique. Cette fonction permet de générer un nombre entre `a` et `b` avec une distribution uniforme.

Ajouter le fichier `random.hpp` dans le dossier `include/local` et créer le test `random.cpp`. Écrire un premier test qui génère 10000 ou 100000 entiers entre 0 et 9 et qui incrémente une `std::map<int, size_t>` pour compter le nombre de fois qu'un entier a été généré. La clé correspond à l'entier généré et la valeur correspond au nombre. Afficher cette `std::map`.

3.1 Quelle est la différence entre `std::map` et `std::unordered_map` ?

3.2 Essayer d'appeler `local::random(0.f, 1.f)`, quelle est l'erreur et pourquoi ?

Écrire la version qui prend des nombres décimaux en utilisant `std::is_floating_point`. Pour l'implémentation de la fonction, on utilisera une `std::uniform_real_distribution` à la place d'une `std::uniform_int_distribution`.

Tester cette version de `local::random` en faisant un test similaire à celui de la version entière. Pour un affichage relativement lisible, vous pouvez générer des flottants entre 0 et 0.1 et réduire leur nombres significatifs à 1 ou 2 : les multiplier par 10 ou 100, les convertir en entier et les reconvertir en flottant pour les multiplier par 10 ou 100.

3.3 Comment modifier notre fonction `local::random` pour prendre un compte les différentes distributions (`uniform`, `bernouilli`, `poisson`, `normal`, `sampling`) ?

4 Place libre (remarques, impressions, fin de réponse...)