TP 5 : CRTP & Variadic Function (Parameter Pack) & Adaptor

Prénom : **Nom :**

Compiler avec le options suivantes (ou équivalentes) :
`-Wall -Wextra -Wconversion -Wsign-conversion -std=c++11` (ou `-std=c++14`) `-pedantic`

1 Curiously Recurring Template Pattern (CRTP)

On cherche à écrire le code suivant :

```
auto p = local::make_unique<mother_t>();
p->fct();
auto p1 = local::make_unique<daughter_t>();
p1->fct();
p = p1->clone();
p->fct();
```

`.fct()` de la classe `mother_t` affiche « `mother_t` » et `.fct()` de la classe `daughter_t` affiche « `daughter_t` ».

1.1 Quel est le type de `p` ? de `p1` ? Quels sont les trois affichages voulus ?

1.2 Que retourne la fonction `.clone()` de `mother_t` ? et `daughter_t` ? Pourquoi ?

1.3 Donner l'implémentation des fonctions `.clone()`.

1.4 Pourquoi une redéfinition de `.clone()` est nécessaire ? Comment générer automatiquement la fonction `.clone()` ?

1.5 Écrire une classe `template cloneable` qui prend deux types (`base_t` et `derived_t`) et qui implémente la fonction `.clone()`. On sait que `derived_t` héritera `cloneable` : c'est-à-dire que `cloneable` peut être caster (avec `static_cast`) en `derived_t`.

1.6 De quelle(s) classe(s) hérite `mother_t` ? et `daughter_t` ?

1.7 Avec ce code, quelle est l'erreur de compilation ?

Résoudre le souci avec le mot clé `using` : http://en.cppreference.com/w/cpp/language/using_declaration

2 Affichage Avec `local::print`

La fonction `printf` du langage C a (au moins) deux inconvénients : il faut préciser le type à afficher et il n'est pas possible de gérer les types utilisateurs.
L'utilisation de l'opérateur << entre un `std::ostream` et ce que l'on veut afficher permet de régler ces soucis.

On souhaite écrire une fonction `local::print` qui permettrait d'écrire ce code :

```
local::print("Use", ' ', std::string("local::print"), " to print: ",
             42, " + ", 21.73, " = ", 42 + 21.73, '\n');
```

Pour écrire une telle fonction, il faudra utiliser un *parameter pack* : http://en.cppreference.com/w/cpp/language/parameter_pack

2.1 Écrire une fonction `local::fprint` qui prend un `std::ostream` et qui "force" l'affichage avec `std::flush` dans le flux.

2.2 Écrire une fonction `local::fprint` qui prend un `std::ostream`, un `T const &` et un *parameter pack* `args_t const & ...`

2.3 Écrire la fonction `local::print`.

3 Une fonction `distance(point_t0 const &, point_t1 const &)` générique

Écrire une classe `template` qui correspond à un point.
Écrire une fonction `template distance` qui prend deux points (qui peuvent être de types différents) et qui retourne la distance entre ces deux points.

3.1 Quel est le type de retour de la fonction `distance` ?

3.2 Essayer votre fonction avec différents types (`std::array`, `std::vector`, `std::tuple`, `hopp::vector2`, `sf::Vector2`, `QPointF` et `wxRealPoint`). Quels sont les solutions qui permettent d'accepter tous ces types différents ?

3.3 Écrire les fonctions libres qui permettent d'acceder à l'abscisse de n'importe quel type de point.

3.4 Écrire une fonction `perimeter` qui prend plusieurs points (*parameter pack*) et qui calcule la somme des distances entre le 1er et le 2ème point, entre le 2ème et le 3ème, ... et entre le 1er et le dernier.

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