C++ 2A
TD/TP 4 – Généricité, STL

1 Héritage simple

1.1 Membres public, protected, private

1. Créer une classe A (.hpp et .cpp) dont la déclaration est la suivante :

```
class A
{
private:
  int x;
  void fct1();
protected:
  int y;
  void fct2();
public:
  int z;
  void fct3();
}
```

2. Créer une classe B (.hpp et .cpp) héritant de la classe A.
3. Créer un fichier `main.cpp` utilisant la classe B.
4. Dans la classe B, tester l'accès aux différents membres de la classe ancètre.
5. Cas particulier : dans la classe A et B, définir une méthode de même prototype. Comment appeler dans B la fonction de l'ancètre A ?

1.2 Héritage public, protected, private

1. Créer un fichier `main.cpp` utilisant les classes A et B.
2. Modifier le type d'héritage de la classe B en le passant de public à protected puis private et tester à chaque fois l'accès aux différents membres.

2 Héritage multiple

1. Créer une classe C avec quelques membres.
2. Créer une classe D héritant de la classe A et C, et utilisant leurs méthodes.
3. Tester votre classe D dans un fichier `main.cpp`
4. Cas particulier : Définir une méthode dans A puis dans C portant le même nom. Comment appeler explicitement la méthode souhaitée à partir de la classe D ?

3 Classe abstraite

Une classe abstraite contient au moins la déclaration d'une méthode virtuelle pure et ne peut donc être instanciée.

1. Transformer la classe A en classe abstraite.
2. Quelle erreur de compilation s'affiche si la classe B n'est pas modifiée ?

4 Interface

Une interface est une classe qui contient des déclarations de méthodes virtuelles pures (i.e. sans code) et aucun autre membre.

1. Transformer la classe C en interface.
2. Quelles modifications doivent être apportées à la classe D ?

5 Mise en pratique

5.1 Bricoleur

On s'intéresse à la modélisation d'un bricoleur qui peut effectuer certaines tâches telles que visser, couper, casser. Chacune de ces tâches s'accomplit à l'aide d'un outil adapté. Par exemple, un tournevis est un outil adapté pour visser, on pourrait donc avoir quelque chose ressemblant à :

```
class Bricoleur {
public :
  void visse(Tournevis t) {
    t.visse();
  }
  ...
};
class Tournevis {
  std::string nom;
public :
  Tournevis(std::string const & nom);
  std::string get_nom();
  void visse() {
    cout<<"Tournevis visse";
  }
  ~Tournevis();
};
```

5.1.1 Implantation

Terminer cette implantation "naïve" en définissant des classes Scie, pour couper, Marteau, pour casser, et en complétant la classe bricoleur.

5.1.2 Extension

Cette implantation ne permet pas de définir un tableau d'outils dans lequel mettre des outils de différents types.

1. Définir une classe abstraite Outil qui sera l'ancètre de tous les outils.
2. Définir également des interfaces OutilCassant, OutilCoupant, etc qu'implanteront les différents outils.

5.1.3 Fabrique à outils

Créer une classe OutilFactory avec une méthode statique create(string name) retournant un nouvel outil en fonction de la chaîne passée en paramètre :

```
Outil* tournevis = OutilFactory::create( "Tournevis" );
```

5.1.4 Boîte à outils

Créer un tableau d'outils à remplir à l'aide de la fabrique et afficher le contenu de votre boîte à outils.

5.2 Magic

On souhaite créer un jeu de cartes de type Magic. L'objectif n'est pas d"implanter complètement le jeu mais de voir les techniques utilisées pour mettre en place ce type de problème.
Les cartes peuvent être différents types dont voici un sous-ensemble : :

- des terrains (Lands) : plaine (Plains), île (Island), marais (Swamp), montagne (Mountain), forêt (Forest).
- des sortilèges (Spells)
  - des créatures (Creatures)
  - des artefacts et enchantements (Artifacts&Enchantments)
  - ...

La base du jeu est de poser des terrains pour générer des unités de mana. Chaque type de terrain génère un type de mana différent. Ensuite des sorts peuvent être invoqués si le bon type et la quantité de mana générée par les terrains sont disponibles.
On souhaite modéliser ces différentes cartes. Les terrains et les sortilèges sont deux types de cartes. Les créatures et les artefacts et enchantements sont des types de sortilèges.

1. Créer une classe abstraite `Card` contenant un attribut `string name` et comprenant un constructeur prenant en paramètre le nom de la carte.
2. Créer une méthode virtuelle pure **`protected`** de prototype :

   ```
   virtual ostream & toStream( ostream & out ) const
   ```

   ajoutant le nom de la carte ou flux passé en paramètre.
3. Surcharger l'opérateur << afin d'appeler la méthode `toStream`. Attention la méthode `toStream` doit rester **`protected`** !
4. Créer une classe `SpellCard` héritant de `Card` et qui doit rester virtuelle pure car on ne doit pas pouvoir l'instancier.
5. Créer une classe `Creature` héritant de `SpellCard`. Le constructeur prendra en paramètre le nom de la carte, le nombre de points d'attaques et de points de défense. Cette classe doit pouvoir être instanciée et devra donc définir les méthodes virtuelles pures héritées.
6. Créer un fichier `main.cpp` contenant le code suivant et tester si cela fonctionne, le résultat de l'affichage devra être "Creature Sorcerer 4/3" :

   ```
   int main()
   {
     Creature sorcerer( "Sorcerer", 4, 3 );

     cout << sorcerer << endl;

     return 0;
   }
   ```