# Construction Objets avancée

Giuseppe Lipari

February 25, 2019

## Instructions

Vous devez rendre sur gitlab le code demandé avec un fichier README.md qui contient :

- Vos noms ;
- Pour chaque question :
  - Si vous avez réussi à coder les fonctionnalités demandées
  - La liste de tests de régression correspondants à la question

## TP 4: Templates

Le but de ce TP est de comprendre la programmation par *templates*.

L'intersection de deux ensembles $A$ et $B$ est l'ensemble $C$ qui contient tous les éléments qui sont présent dans le deux ensembles:

$$C = A \cap B = x | x \in A \wedge x \in B$$

L'union de deux ensembles $A$ et $B$ est l'ensemble $C$ qui contient tous les éléments qui sont présent dans un de deux ensembles:

$$C = A \cup B = x | x \in A \vee x \in B$$

L'ensemble $C$ ne contient pas de doublons.

L'objectif final est d'implanter ces deux fonctions de manière la plus générale possible : les fonctions doivent marcher sur n'importe quel container, et sur n'importe quel type de donnée contenue.

### Question 1: vecteurs d'entiers

Écrire deux fonctions, `set_intersection` et `set_union` que, à partir de deux vecteurs d'entiers, remplissent un vecteur qui contient l'intersection (l'union, respectivement) de deux vecteurs d'entiers.

Voici le prototype de la fonction `set_intersection()` :

```
void set_intersection(std::vector<int>::const_iterator a_begin,
        std::vector<int>::const_iterator a_end,
        std::vector<int>::const_iterator b_begin,
        std::vector<int>::const_iterator b_end,
        std::back_insert_iterator<std::vector<int>> c_begin);
```

La fonction sera utilisé comme dans le programme suivant:

```
vector<int> vec_a = {1, 2, 3, 4};
vector<int> vec_b = {3, 4, 5, 6};
vector<int> vec_c;

set_intersection(begin(vec_a), end(vec_a), begin(vec_b), end(vec_b),
                 back_inserter(vec_c));
```

La fonction `set_union` a le même prototype, et on l'utilise de la même manière.

1. Testez les deux fonctions, surtout dans les cas limites (intersection nulle, l'union de deux ensembles vides, etc.)

## Question 2: Généralisation sur les containers

1. Généralisez les fonctions développé dans la question 2 en utilisant des templates.
2. Testez les fonctions sur des vecteurs et des listes d'entiers.

## Question 3: Containers d'objets

1. Créer de vecteurs et des listes d'objets de type `MyClass`, et essayez de faire des intersections et des unions avec les fonctions développées dans la Question 3. Vérifiez que tout est correct.
2. Quel sont les caracteristiques minimales de la classe pour pouvoir faire l'intersection et l'union ?

## Question 4

1. Est-ce qu'on peut appeler `set_intersection` sur des containers qui contiennent objets de type différents ?
2. Si non, pourquoi ? Si oui, quel sont le conditions minimales sur les types des objets, et quel est le résultat finale ?

## Question 5

1. Essayez d'appliquer vos fonctions sur des `map<int, string>`.
2. Est-ce que ça marche ? Si oui, pourquoi ? Si non, pourquoi ?

## Question 6

1. Généraliser la fonction `set_intersection` avec un paramétre template additionnel `F` qui represents une fonction de comparaison entre objets des deux containers.

2. Appliquer la fonction `set_intersection` sur une `map<int, string>` and sur un `vector<int>`, le resultat sera produit sur une `map<int, string>` :

```
bool my_compare(const std::pair<int, string> &x, int y);

map<int, string> a, b;
vector<int> v = {1, 3, 5};

a[1] = "A"; a[2] = "B"; a[3] = "C"; a[4] = "D";

set_intersection(begin(a), end(a), begin(v), end(v),
        my_compare, inserter(b));

// b should contain (1, "A") and (3, "C") and nothing else.
```

3. Testez le résultat.