Université Paris Diderot **Langages à objets avancés**
M1 **Année 2019-2020**

TP 9 : Templates

Important Examen : l'examen se fera sur les machines de l'UFR, il vous est donc recommandé de tester si votre environnement fonctionne et repérer comment utiliser l'IDE qui vous convient le plus, s'il est configuré correctement pour C++...

Exercice 1 Pile générique

On utilisera comme point de départ pour implémenter une pile générique le code suivant :

```
template <typename T> class Pile
{
    private:
    int size;
    int top;
    T* stacktab;
    .....
};
```

1. Écrivez un constructeur vide et un autre avec comme argument la taille, des méthodes pour tester si la pile est vide, si elle est pleine, une méthode pour empiler (seulement lorsqu'il y a de la place, ne rien faire sinon) et une pour dépiler (on suppose qu'il y a bien des éléments).
2. Pour permettre un affichage simplifié, vous redéfinirez également l'opérateur `<<` avec la technique "opérateur comme fonction ordinaire". Si vous avez besoin de déclarer cette fonction comme amie de la classe `Pile` vous pourrez le faire en ajoutant cette déclaration à la classe :

   ```
   template<typename U>
   friend std::ostream& operator<<(std::ostream&, const
      Pile<U>&);
   ```

3. Testez avec `Pile<int>` et `Pile<char>`.
4. On veut maintenant empiler dans la pile des Fractions de l'exercice précédent, mais Fraction n'a pas de constructeur par défaut. Que peut-on empiler sans ajouter de constructeur par défaut à Fraction. Écrivez quelques tests et faites le nécessaire pour que l'affichage de la pile de Fractions soit convenable.

Exercice 2 Valeur admises génériques et surcharges d'opérateurs

On reprend l'exercice 2 du TP précédent.

1. Redéfinissez l'interface en remplaçant `char` par un type `T` générique. Quelle modification (*très simple !*) devez vous apporter à vos classes `Intervalle` et `TableauValeurs` pour qu'elles continuent de fonctionner comme avant ?
2. Maintenant, je veux rendre ces deux dernières classes génériques aussi. Quelles condition doit remplir le type choisi pour remplacer `T` ?
3. On veut maintenant ajouter à l'interface l'addition et la soustraction d'un entier. L'idée est de retourner un nouvel intervalle où on a décalé les valeurs admises de la valeur de l'entier. Là encore on restreint les types possibles. Testez avec `int` et `char`. pour les tests avec `char`, assurez-vous que vous restez dans la partie des caractères imprimables.

S'il vous reste du temps

Faites en priorité l'exercice 3 du TP précédent, sinon faites le suivant.

Exercice 3 On définit maintenant une classe abstraite `Element` qui prendra comme sous-classes `Operateur` et `Nombre`, les sous-classes de `Operateur` seront `Plus` et `Moins` (binaire), et pour `Nombre` on définira une seule sous-classe `Fraction` (On modifiera en conséquence le code précédent). Les opérateurs redéfiniront l'opérateur `()` avec deux arguments de type `Fraction`.

Écrivez maintenant, en utilisant la pile définie dans le premier exercice, une fonction `Fraction calculFraction(Element ** tab` int size), qui calcule le résultat de l'expression postfixée contenue dans le tableau en supposant que tous les `Nombre` seront des `Fraction`. Cette fonction sera mise dans le même fichier que le `main`.

Par exemple :

```
calcul([1/2; 5/2; 1/2; Moins; Plus]) == 5/2
```

Écrivez une version générique de ces classes (fonctionnant avec tout type de nombre redéfinissant les opérateurs `+` et `-`).

Aide pour la programmation

— Pour pouvoir définir l'opérateur `ostream & operator<<(ostream \& o` const Element & f), et faire en sorte qu'il marche avec tout type d'élement, on déclarera une méthode virtuelle pure `string toString()` qui aura le même sens qu'en Java et sera à redéfinir quand nécessaire.
— Pour la fonction `calculFraction`, il faut faire une pile de pointeur de `Fraction` et utiliser la fonction `dynamic\_cast<>` quand c'est nécessaire.
— Si on veut faire d'autres sous-classes de `Nombre`, il faudra surcharger les opérateurs `()` en conséquence.