Compte Rendu TP4 CPOO

Théo CHAPON, Hassan EL OMARI ALAOUI

INSA de Rennes
4INFO, groupe 1.1

19 octobre 2014

TP4 : Création et utilisation d'un template

1 Etat du TP

Le projet est terminé, fonctionel et testé.

2 Objectif

Ce TP avait pour but d'apprendre à utiliser et créer une classe template. Dans notre cas, il s'agit d'une classe template "Ensemble" qui utilise la classe template "List".

3 Réalisation

Nous devions réaliser un template permettant de représenter un ensemble d'objets non trié et sans doublon. Pour gérer les ensembles, nous devions surchager les opérateurs arithmétiques de base :

- '+' pour l'union de deux ensembles
- '*' pour l'intersection de deux ensembles
- '-' pour la soutraction de deux ensembles
- '/' pour la différence de deux ensembles
- '«' pour gérer l'affichage des ensembles
- '»' pour gérer la création par fichier d'un ensemble

Nous devions aussi produire un constructeur par recopie. La totalité des fonctions du template "Ensemble" ont utilisé des méthodes provenant de la classe "List".

Une seul petite contrainte a été rencontrée dans ce TP, c'est l'utilisation d'une énumération de la classe "List" protéger par protected. Cela nécessite de faire de l'héritage mais ça complique le code de la classe "Ensemble".

Listing 1 – Définition de la class template Ensemble

```
#pragma once
#include "stdafx.h"
#include "list.h"

using namespace std;
template <class T>
// == operator required
class Ensemble {
private:
  List<T> _ensemble;/*<! Set >*/
  /**
   * \fn ostream& _print(ostream& os)
   * \brief Display Ensemble object
   * \param[in,out] os output stream
   * \return ostream& reference of output stream
   */
  ostream& _print(ostream& os) {
    return os << _ensemble;
  }
public:
  /**
  * \fn Ensemble<T>()
  * \brief Ensemble<T> default constructor
  */
  Ensemble<T>():_ensemble() {}
  /**
  * \fn Ensemble<T>(const Ensemble<T>& e)
  * \brief Ensemble<T> copy constructor
  * \param[in] e : Ensemble
  */
  Ensemble<T>(const Ensemble<T>& e):_ensemble(e._ensemble) {}

  /**
  * \fn bool exists(T value)
  * \brief Test whether value exists in the set or not
  * \param[in] value : T
  * \return Return true if value exist in the set, false otherwise
  */
```

```
39   bool exists(T value) {
40     for (ListIterator<T> it = _ensemble.beg(); !it.finished(); ++it){
41       if (it.get() == value)
42       {
43         return true;
44       }
45     }
46     return false;
47   }
48
49   /**
50   * \fn friend Ensemble<T>& operator+=(Ensemble<T>& e1, Ensemble<T> e2)
51   * \brief Overloads += operator
52   * Return the union between two sets
53   * \param[in,out] e1 Ensemble<T>&
54   * \param[in] e2 Ensemble<T>&
55   * \return Ensemble<T>&
56   */
57   friend Ensemble<T>& operator+=(Ensemble<T>& e1, Ensemble<T>& e2) {
58     for (ListIterator<T> it = e2._ensemble.beg(); !it.finished(); ++it)
59     {
60       if (!e1.exists(it.get()))
61       {
62         e1._ensemble.addElement(it.get()/*, List<T>::LP_last*/); // we can't
               access LP_last beacause it's protected
63       }
64     }
65     return e1;
66   }
67
68   /**
69   * \fn friend Ensemble<T> operator+(const Ensemble<T>& e1, Ensemble<T> e2)
70   * \brief Overloads + operator
71   * Return the union between two sets
72   * \param[in,out] e1 Ensemble<T>&
73   * \param[in] e2 Ensemble<T>&
74   * \return Ensemble<T>
75   */
76   friend Ensemble<T> operator+(const Ensemble<T>& e1, Ensemble<T>& e2) {
77     Ensemble<T> e3(e1);
78     return e3 += e2;
79   }
80
81   /**
82   * \fn friend Ensemble<T>& operator-=(Ensemble<T>& e1, Ensemble<T> e2)
83   * \brief Overloads -= operator
84   * Return the substruction between two sets
85   * \param[in,out] e1 Ensemble<T>&
86   * \param[in] e2 Ensemble<T>&
```

```
87    * \return Ensemble<T>&
88    */
89    friend Ensemble<T>& operator-=(Ensemble<T>& e1, Ensemble<T>& e2){
90      for (ListIterator<T> it = e2._ensemble.beg(); !it.finished(); ++it)
91      {
92        e1._ensemble.delElement(it.get());
93      }
94      return e1;
95    }
96
97    /**
98    * \fn friend Ensemble<T> operator-(const Ensemble<T>& e1, Ensemble<T> e2)
99    * \brief Overloads - operator
100   * Return the substruction between two sets
101   * \param[in,out] e1 Ensemble<T>&
102   * \param[in] e2 Ensemble<T>&
103   * \return Ensemble<T>
104   */
105   friend Ensemble<T> operator-(const Ensemble<T>& e1, Ensemble<T>& e2){
106     Ensemble<T> e3(e1);
107     return e3 -= e2;
108   }
109
110   /**
111   * \fn friend Ensemble<T>& operator*=(Ensemble<T>& e1, Ensemble<T> e2)
112   * \brief Overloads *= operator
113   * Return the intersection between two sets
114   * \param[in,out] e1 Ensemble<T>&
115   * \param[in] e2 Ensemble<T>&
116   * \return Ensemble<T>&
117   */
118   friend Ensemble<T>& operator*=(Ensemble<T>& e1, Ensemble<T>& e2) {
119     Ensemble e3(e1);
120     for (ListIterator<T> it = e3._ensemble.beg(); !it.finished(); ++it)
121     {
122       if (!e2.exists(it.get()))
123       {
124         e1._ensemble.delElement(it.get());
125       }
126     }
127     return e1;
128   }
129
130   /**
131   * \fn friend Ensemble<T> operator*(const Ensemble<T>& e1, Ensemble<T> e2)
132   * \brief Overloads * operator
133   * Return the intersection between two sets
134   * \param[in,out] e1 Ensemble<T>&
135   * \param[in] e2 Ensemble<T>&
```

```
136   * \return Ensemble<T>
137   */
138   friend Ensemble<T> operator*(const Ensemble<T>& e1, Ensemble<T>& e2) {
139     Ensemble<T> e3(e1);
140     return e3 *= e2;
141   }
142
143   /**
144   * \fn friend Ensemble<T>& operator/=(Ensemble<T>& e1, Ensemble<T> e2)
145   * \brief Overloads /= operator
146   * Return the difference between two sets
147   * \param[in,out] e1 Ensemble<T>&
148   * \param[in] e2 Ensemble<T>&
149   * \return Ensemble<T>&
150   */
151   friend Ensemble<T>& operator/=(Ensemble<T>& e1, Ensemble<T>& e2){
152     return e1 = (e1 + e2) - (e1 * e2);
153   }
154
155   /**
156   * \fn friend Ensemble<T> operator/(const Ensemble<T>& e1, Ensemble<T> e2)
157   * \brief Overloads / operator
158   * Return the difference between two sets
159   * \param[in,out] e1 Ensemble<T>&
160   * \param[in] e2 Ensemble<T>&
161   * \return Ensemble<T>
162   */
163   friend Ensemble<T> operator/(const Ensemble<T>& e1, Ensemble<T>& e2){
164     Ensemble<T> e3(e1);
165     return e3 /= e2;
166   }
167
168   /**
169   * \fn template<typename T> friend istream& operator>>(istream& os, Ensemble<T>&
        f)
170   * \brief Overloads >> operator
171   * Read Ensemble object
172   * \param[in,out] os istream&
173   * \param[in] f Ensemble&
174   * \return istream&
175   */
176   template<typename T> friend istream& operator>>(istream& is, Ensemble<T>& e);
177
178   /**
179   * \fn template<typename T> friend ostream& operator<<(ostream& os, Ensemble<T>&
        f)
180   * \brief Overloads << operator
181   * Return output stream to display Ensemble object
182   * \param[in,out] os ostream&
```

```
183   * \param[in] f Ensemble&
184   * \return ostream&
185   */
186   template<typename T> friend ostream& operator<<(ostream& os, Ensemble<T>& e);
187 
188 };
189 template <class T>
190 istream& operator>>(istream& is, Ensemble<T>& e){
191   return is >> e._ensemble;
192 }
193 template <class T>
194 ostream& operator<<(ostream& os, Ensemble<T>& e) {
195   return e._print(os);
196 }
```