Compte Rendu TP5 CPOO

Maxime Havez, Gareth Thiveux

INSA de Rennes
4INFO, groupe 2.2

4 novembre 2010

Listing 1 – Classe template Ensemble

```
1  //=========================================================================
2
3  // Fichier    :  ensemble.h
4
5  // Description:  Classe template Ensemble
6
7  // Auteurs    :  Maxime HAVEZ ; Gareth THIVEUX
8
9  // Date       :  04/11/10
10
11 //=========================================================================
12
13
14
15 #ifndef ENSEMBLE_H
16
17 #define ENSEMBLE_H
18
19
20
21 #include "list.h"
22
23 #include <iostream>
24
25
26
27
28
29 // ========================================================================
30 // Classe: definition de la classe Ensemble<T> decrivant des ensembles
```

```
31 // ========================================================================
32
33 template <class T> class Ensemble {
34
35   /*  On choisit une relation d'agregation entre List et Ensemble.
36
37   On cree un ensemble qui ne possede pas les memes proprietes que les listes.
38
39   On n'offre pas a l'ensemble les methodes que possede la liste.
40
41
42
43   Par contre comme la liste est le conteneur, on va utiliser les methodes de la
       classe List
44
45   pour definir les methodes que l'on offre a la classe Ensemble.  */
46
47
48
49 private :
50
51   // La liste
52
53   List<T> liste;
54
55
56
57 public :
58
59 // ========================================================================
60 // But: constructeur par recopie
61 // ========================================================================
62
63   Ensemble(const Ensemble<T>& E):liste(E.liste){}
64
65
66
67   /*  On retourne l'ensemble qui est construit dans la fonction !
68
69   On ne doit pas rendre de reference dans ce cas la ! */
70
71
72
73 // ========================================================================
74 // But: constructeur par defaut
75 // ========================================================================
76
77   Ensemble(){}
78
```

```
79
80
81 // ========================================================================
82 // But: destructeur
83 // ========================================================================
84
85   ~Ensemble(){}
86
87
88
89
90
91 //========================================================================
92 // Methode   :  bool appartient(const T elem) const
93 // But       :  Teste l'appartence d'un element a l'ensemble courant
94 // Paramtres:   elem  : reference constante vers un element
95 // Retour    :  booleen
96 //========================================================================
97
98   bool appartient(const T& elem) const {
99
100    /*
101
102      bool res=false;
103
104      ListIterator<T> it = liste.beg();//it?rateur sur la liste courante
105
106      for(;!it.finished();++it){
107
108        if (it.get() == elem) res=true;
109
110      }
111
112      return res;
113
114    */
115
116
117
118    //En utilisant la fonction == de liste
119
120    return ((liste==elem) != 0 );
121
122  }
123
124
125
126 //========================================================================
127 // Methode   :  Ensemble<T> operator+(const T& elem) const
```

```
128 // But        :  Ajout d'un element a l'ensemble
129 // Paramtres:   elem  : reference constante vers un element
130 // Retour     :  Ensemble<T>
131 //=====================================================================
132
133   Ensemble<T> operator+(const T& elem) const {
134
135     Ensemble<T> tmp(*this);
136
137     tmp.liste = tmp.liste + elem;
138
139     return tmp;
140
141   }
142
143
144
145 //=====================================================================
146 // Methode    :  Ensemble<T> operator+(Ensemble<T>& E) const
147 // But        :  Union de deux ensembles
148 // Paramtres:   E : reference vers un ensemble
149 // Retour     :  Ensemble<T>
150 //=====================================================================
151
152   Ensemble<T> operator+(Ensemble<T>& E) const {
153
154     Ensemble<T> tmp(*this); // Necessite d'avoir le constructeur par recopie
155
156     ListIterator<T> it = E.liste.beg(); //iterateur sur la liste de E
157
158     for(;!it.finished();++it){
159
160       if(!(tmp.appartient(it.get()))) {tmp.liste = tmp.liste + it.get() ;}
161
162     }
163
164     return tmp;
165
166   }
167
168
169
170 //=====================================================================
171 // Methode    :  Ensemble<T> operator/(Ensemble<T>& E) const
172 // But        :  Difference ensembliste de deux ensembles
173 // Paramtres:   E : reference vers un ensemble
174 // Retour     :  Ensemble<T>
175 //=====================================================================
176
```

```
177   Ensemble<T> operator/(Ensemble<T>& E) const {
178
179     Ensemble<T> tmp(*this);
180
181     ListIterator<T> it=E.liste.beg();
182
183     for(;!it.finished();++it){
184
185       if((tmp.appartient(it.get()))){tmp.liste = tmp.liste - it.get() ;}
186
187       else{tmp.liste = tmp.liste + it.get() ;}
188
189     }
190
191     return tmp;
192
193   }
194
195
196
197 //========================================================================
198 // Methode    :  Ensemble<T> operator*(Ensemble<T>& E) const
199 // But        :  Intersection de deux ensembles
200 // Paramtres:    E : reference vers un ensemble
201 // Retour     :  Ensemble<T>
202 //========================================================================
203
204   Ensemble<T> operator*(Ensemble<T>& E) const {
205
206     Ensemble<T> tmp;
207
208     ListIterator<T> it=E.liste.beg();
209
210     for(;!it.finished();++it){
211
212       if((this->appartient(it.get()))){tmp.liste = tmp.liste + it.get() ;}
213
214     }
215
216     return tmp;
217
218   }
219
220
221
222 //========================================================================
223 // Methode    :  Ensemble<T> operator-(Ensemble<T>& E) const
224 // But        :  Soustraction d'un ensemble par un autre
225 // Paramtres:    E : reference vers un ensemble
```

```
226 // Retour    :  Ensemble<T>
227 //============================================================================
228
229   Ensemble<T> operator-(Ensemble<T>& E) const {
230
231     Ensemble<T> tmp(*this);
232
233     ListIterator<T> it=E.liste.beg();
234
235     for(;!it.finished();++it){
236
237       if((tmp.appartient(it.get()))){tmp.liste = tmp.liste - it.get();}
238
239     }
240
241     return tmp;
242
243   }
244
245
246
247 // ============================================================================
248
249 // Pre-declaration : operateur d'affichage d'une liste dans un flux
250
251 // ============================================================================
252
253   template <class S>/*pas forcement celui de la classe*/ friend std::ostream&
        operator<<(std::ostream& out,const Ensemble<S>& lref);
254
255
256
257 // ============================================================================
258
259 // Pre-declaration : operateur de lecture d'une liste dans un flux
260
261 // ============================================================================
262
263   template <class S> friend std::istream& operator>>(std::istream&
        in,Ensemble<S>& lref);
264
265
266
267 };
268
269
270
271
272
```

```
273 //=========================================================================
274 // Methode   :  std::ostream& operator<<(std::ostream& out,const Ensemble<T>&
    lref)
275 // But       :  operateur d'affichage d'une liste dans un flux
276 // Paramtres:   out : flux ;  lref : reference vers un ensemble
277 // Retour    :  std::ostream&
278 //=========================================================================
279
280   template <class T>
281
282   std::ostream& operator<<(std::ostream& out,const Ensemble<T>& lref) {
283
284     out << lref.liste ;// affichage du cardinal est deja gere dans la liste
285
286     return out;
287
288   }
289
290
291
292 //=========================================================================
293 // Methode   :  std::istream& operator>>(std::istream& in,Ensemble<T>& lref)
294 // But       :  operateur de lecture d'une liste dans un flux
295 // Paramtres:   in  : flux ;  lref : reference vers un ensemble
296 // Retour    :  std::istream&
297 //=========================================================================
298
299   template <class T>
300
301   std::istream& operator>>(std::istream& in,Ensemble<T>& lref){
302
303     in >> lref.liste ;
304
305     return in;
306
307   }
308
309
310
311 #endif
```

Listing 2 – Classe template List

```
1 //=========================================================================
2 // list.h
3 //=========================================================================
4
5 #ifndef LIST_H
6 #define LIST_H
7
```

```cpp
#include <iostream>

// Pre-declarations necessaires pour l'utilisation des classes amies
template <class T> class List;
template <class T> class ListIterator;

// ============================================================================
// Classe: definition de la classe ListElement<T> decrivant un element de
//         liste.
//         Tous les membres de la classe ListElement sont definis prives,
//         ainsi seules les classes amies List et ListIterator y ont accs.
// ============================================================================
template <class T> class ListElement {
  private:
    // Valeur de l'element
    T _value;

    // Membres pour le chainage
    ListElement<T>* _prev;
    ListElement<T>* _next;

    // Constructeurs et destructeur : ils mettent a jour le chainage au sein de
       la liste
    ListElement(const T& v) : _value(v), _prev(0), _next(0) {}
    ListElement(ListElement<T>* p, ListElement<T>* n)
      :  _prev(p), _next(n) {
      if (n != 0) n->_prev = this;
      if (p != 0) p->_next = this;
    }
    ListElement(const T& v,ListElement<T>* p, ListElement<T>* n)
      : _value(v), _prev(p), _next(n) {
      if (n != 0) n->_prev = this;
      if (p != 0) p->_next = this;
    }
    ~ListElement() {
      if (_prev != 0) _prev->_next = _next;
      if (_next != 0) _next->_prev = _prev;
    }

    // Classes amies
    friend class List<T>;
    friend class ListIterator<T>;
};


// Predeclaration de List pour pouvoir pre-declarer des operateurs
template <class T> class List;

```

```
56 // Pre-declaration pouvoir declarer ces operateurs amis de List
57 template <class T> std::ostream& operator<<(std::ostream& out, const List<T>&
       lref);
58 template <class T> std::istream& operator>>(std::istream& in, List<T>& lref);
59
60 // ========================================================================
61 // Classe: definition de la classe List<T> decrivant des listes generiques
62 //         La gestion de la liste est en double chainage avec deux elements
63 //         fictifs _head et _tail pour gerer le chainage.
64 // ========================================================================
65 template <class T> class List {
66   private:
67     // Les deux elements fictifs
68     ListElement<T>* _head;
69     ListElement<T>* _tail;
70
71     // Nombre d'elements contenus
72     int _card;
73
74     // ========================================================================
75     // But: suppression des elements contenus (mais pas les elements fictifs)
76     // ========================================================================
77     void _freelist() {
78       ListElement<T>* tmp = _head->_next;
79       while (tmp != _tail) {
80         ListElement<T>* n = tmp->_next;
81         _card--;
82         delete tmp;
83         tmp = n;
84       }
85     }
86
87   protected:
88     // Definition du type enumere eListPosition
89     // Definition des differents emplacements d'insertion d'un nouvel element
90     // dans une liste.
91     enum eListPosition {
92       LP_first = -2, // en debut de liste
93       LP_last = -1,  // en fin de liste
94       LP_pos = 0     // a une position donnee (entre 1 et le cardinal de la liste)
95     };
96
97   public:
98     // ========================================================================
99     // But: constructeur par defaut : creation d'une liste vide
100    // ========================================================================
101    List() : _card(0) {
102      _head = new ListElement<T>(0, 0);
103      _tail = new ListElement<T>(_head, 0);
```

```
104       _head->_next = _tail;
105     }
106
107     // ========================================================================
108     // But: constructeur par recopie
109     // ========================================================================
110     List(const List<T>& lref)
111       : _card(0) {
112       _head = new ListElement<T>(0, 0);
113       _tail = new ListElement<T>(_head, 0);
114       _head->_next=_tail;
115
116       ListElement<T>* tmp = lref._head->_next;
117       while (tmp != lref._tail) {
118         ListElement<T>* toadd;
119         // Ajout a la fin de la liste
120         toadd = new ListElement<T>(tmp->_value,_tail->_prev,_tail);
121         _card++;
122         tmp = tmp->_next;
123       }
124     }
125
126     // ========================================================================
127     // But: destructeur
128     // ========================================================================
129     ~List() {
130       _freelist();
131       delete _head;
132       delete _tail;
133     }
134
135     // ========================================================================
136     // But: operateur d'affectation
137     // ========================================================================
138     List<T>& operator=(const List<T>& lref) {
139       if (this != &lref) {
140         _freelist();
141         ListElement<T>* tmp = lref._head->_next;
142         while (tmp != lref._tail) {
143     ListElement<T>* toadd;
144     // Ajout a la fin de la liste
145     toadd = new ListElement<T>(tmp->_value,_tail->_prev,_tail);
146     _card++;
147     tmp = tmp->_next;
148         }
149       }
150       return *this;
151     }
152
```

```
153     // ============================================================================
154     // But: test d'appartenance d'un element a une liste, rend l'index si present
155     // ============================================================================
156     bool operator==(const T& v) const {
157       int res = 0;
158       int idx = 0;
159       ListElement<T>* tmp = _head->_next;
160       while ((tmp != _tail) && (!res)) {
161         res = (tmp->_value == v);
162         tmp = tmp->_next;
163         idx++;
164       }
165       return (res ? idx : 0);
166     }
167 
168     // ============================================================================
169     // But: cardinal de la liste
170     // ============================================================================
171     int card() const { return _card; }
172 
173     // ============================================================================
174     // But: ajout d'un element a une liste (par defaut en tte de liste)
175     // ============================================================================
176     void addElement(const T& v, eListPosition pos = LP_first) {
177       ListElement<T>* toadd;
178       switch (pos) {
179       case LP_first:  // Ajout en debut
180         toadd = new ListElement<T>(v, _head, _head->_next);
181         _card++;
182         break;
183       case LP_last: // Ajout en fin
184         toadd = new ListElement<T>(v, _tail->_prev, _tail);
185         _card++;
186         break;
187       case LP_pos: // Pas d'ajout en position 0 de la liste
188       default:
189         int realpos = pos;
190         ListElement<T>* tmp = _head->_next;
191         while ((tmp != _tail) && (realpos > 1)) {
192     tmp = tmp->_next;
193     realpos--;
194         }
195         // Ajout effectif de l'element
196         if (realpos == 0) {
197     toadd = new ListElement<T>(v,tmp->_prev,tmp);
198         }
199         _card++;
200         break;
201       }
```

```
202     }
203
204     // ========================================================================
205     // But: operateur d'ajout d'un element a une liste (l'ajout se fait en
206     //    debut de liste)
207     // ========================================================================
208     List<T> operator+(const T& v) const {
209       List<T> lres(*this);
210       lres.addElement(v,LP_first);
211       return lres;
212     }
213
214     // ========================================================================
215     // But: operateur de suppression d'un element d'une liste
216     // ========================================================================
217     List<T> operator-(const T& v) const {
218       List<T> lres(*this);
219       lres.delElement(v);
220       return lres;
221     }
222
223     // ========================================================================
224     // But: suppression d'un element
225     // ========================================================================
226     void delElement(const T& v) {
227       // Recherche de l'element
228       ListElement<T>* tmp = _head->_next;
229       while ((tmp != _tail) && (tmp->_value != v)) {
230         tmp = tmp->_next;
231       }
232       // Si l'element a ete trouve, le detruire
233       if (tmp != _tail) {
234         delete tmp;
235         _card--;
236       }
237     }
238
239     // ========================================================================
240     // But: accs a un element donne de la liste en donnant un indice
241     // ========================================================================
242     T& operator[](const int& idx) const {
243       int id = 1;
244       ListElement<T>* tmp = _head->_next;
245       while ((tmp != _tail) && (id != idx)) {
246         tmp = tmp->_next;
247         id++;
248       }
249       return tmp->_value;
250     }
```

```
251
252     // ========================================================================
253     // But: iterateur de liste a partir du debut
254     // ========================================================================
255     ListIterator<T> beg() const {
256       return ListIterator<T>(*this);
257     }
258
259     // ========================================================================
260     // But: iterateur de liste a partir de la fin
261     // ========================================================================
262     ListIterator<T> end() {
263       ListIterator<T> res(*this);
264       res._crtelt = _tail->_prev;
265       return res;
266     }
267
268     // ========================================================================
269     // But: iterateur de liste a partir d'une position donnee
270     // ========================================================================
271     ListIterator<T> pos(const int& idx) {
272       int realpos(idx);
273       ListIterator<T> res(*this);
274       ListElement<T>* tmp = _head->_next;
275       while ((tmp != _tail) && (realpos > 1)) {
276         tmp = tmp->_next;
277         realpos--;
278       }
279       res._crtelt = tmp;
280       return res;
281     }
282
283     // ========================================================================
284     // But: operateur d'affichage d'une liste dans un flux
285     // ========================================================================
286     friend std::ostream& operator<< <T>(std::ostream& out,const List<T>& lref);
287
288     // ========================================================================
289     // But: operateur de lecture d'une liste dans un flux
290     // ========================================================================
291     friend std::istream& operator>> <T>(std::istream& in,List<T>& lref);
292
293     // Classe amie
294     friend class ListIterator<T>;
295 };
296
297
298
299 // ============================================================================
```

```
300 // Classe: definition de la classe ListIterator<T> de parcours des listes
301 // ========================================================================
302 template <class T> class ListIterator {
303     // La liste de reference
304     const List<T>& _listref;
305     // La position courante
306     ListElement<T>* _crtelt;
307 
308     // Constructeur : on ne cree un iterateur que grce a la classe List
309     ListIterator(const List<T>& lref) : _listref(lref) { _crtelt =
        lref._head->_next; }
310 
311   public:
312     // Destructeur */
313     ~ListIterator() {}
314     // Fin du parcours (on est sur l'un des elements fictifs
315     int finished() const { return ((_crtelt == _listref._tail) || (_crtelt ==
        _listref._head)); }
316     // Parcours en marche avant
317     ListIterator<T>& operator++() { _crtelt = _crtelt->_next; return *this; }
318     // Parcours en marche arrire
319     ListIterator<T>& operator--() { _crtelt = _crtelt->_prev; return *this; }
320     // Recuperation de l'element courant
321     T& get() { return _crtelt->_value; }
322 
323     // Classe amie
324     friend class List<T>;
325 };
326 
327 template <class T>
328 std::ostream& operator<<(std::ostream& out, const List<T>& lref) {
329   out << lref.card() << "    ";
330   for (ListIterator<T> iterlst = lref.beg(); !(iterlst.finished()); ++iterlst) {
331     out << iterlst.get() << "  ";
332   }
333   return out;
334 }
335 
336 template <class T>
337 std::istream& operator>>(std::istream& in, List<T>& lref) {
338   int nb;
339   in >> nb;
340   for (int i = 0; i < nb; i++) {
341     T tmp;
342     in >> tmp;
343     lref.addElement(tmp, List<T>::LP_last);
344   }
345   return in;
346 }
```

```

#endif // LIST_H
```

Listing 3 – Main

```
//=========================================================================
// main.cpp
//=========================================================================

#include "ensemble.h"
#include <fstream>
#include <iostream>
#include <string>

//=========================================================================
// Methode    :  Ensemble<int> lire(char* fname)
// But        :  Lecture d'un ensemble dans un fichier texte
// Paramtres:    fname :
// Retour     :  Ensemble<int>
//=========================================================================
Ensemble<int> lire(const std::string& fname) {
  Ensemble<int> res;
  // Ouverture du fichier contenant l'ensemble
  std::ifstream input(fname.c_str(), std::ios::in);

  if (!input) {
    std::cerr  << "Erreur de lecture de " << fname << std::endl;
  }
  else {
    input >> res;
    std::cout << "Contenu du fichier \"" << fname << "\" = " << res << std::endl;
  }

  // Fermeture du fichier
  input.close();

  return res;
}

//=========================================================================
// Methode    :  void main()
// But        :  Programme principal de test
// Paramtres:    -
//=========================================================================
int main() {
  // Lecture des ensembles
  Ensemble<int> e1 = lire("test1.txt");
  Ensemble<int> e2 = lire("test2.txt");

  // Affichage des ensembles
```

```
46   std::cout << "e1 = " << e1 << std::endl;
47   std::cout << "e2 = " << e2 << std::endl;
48
49   // Operations sur les ensembles
50   std::cout << "Union        :: " << (e1 + e2) << std::endl;
51   std::cout << "Intersection :: " << (e1 * e2)<< std::endl;
52   std::cout << "Soustraction :: " << (e1 - e2)<< std::endl;
53   std::cout << "Difference   :: "   << (e1 / e2)<< std::endl;
54   return 0;
55 }
56
57 //=========================================================================
58 // Traces d'execution - Resultats de tests :
59 /*
60 Contenu du fichier "test1.txt" = 10    1  2  3  4  5  6  7  8  9  10
61
62 Contenu du fichier "test2.txt" = 5    1  16  8  23  6
63
64 e1 = 10    1  2  3  4  5  6  7  8  9  10
65
66 e2 = 5    1  16  8  23  6
67
68 Union        :: 12    23  16  1  2  3  4  5  6  7  8  9  10
69
70 Intersection :: 3    6  8  1
71
72 Soustraction :: 7    2  3  4  5  7  9  10
73
74 Difference   :: 9    23  16  2  3  4  5  7  9  10
75
76 */
77
78 //=========================================================================
```