TP4 : Manipulation d'ensembles génériques

Paul Chaignon, Xavier Fraboulet

INSA de Rennes
4INFO, groupe 2.2

16 octobre 2013

Listing 1 – Ensemble.h

```
#ifndef ENSEMBLE_H
#define ENSEMBLE_H

#include "List.h"

template <class T> class Ensemble: public List<T> {
public:
  Ensemble<T> operator+(const Ensemble<T>& e) const {
    Ensemble<T> result(*this);
    result.addAllElements(e);
    return result;
  }

  Ensemble<T> operator-(const Ensemble<T>& e) const {
    Ensemble<T> result(*this);
    result.delAllElements(e);
    return result;
  }

  Ensemble<T> operator/(const Ensemble<T>& e) const {
    return (*this - e) + (e - *this);
  }

  Ensemble<T> operator*(const Ensemble<T>& e) const {
    Ensemble<T> result;
    ListIterator<T> it = e.beg();
    while(!it.finished()) {
      T el = it.get();
      if(*this == el) {
        result.addElement(el);
```

```
31        }
32        it++;
33      }
34      return result;
35    }
36
37    virtual void addElement(const T& e, eListPosition pos = LP_first) {
38      if(!(*this == e)) {
39        List::addElement(e);
40      }
41    }
42 };
43
44 #endif
```

Listing 2 – List.h

```
1  //=========================================================================
2  // list.h
3  //=========================================================================
4
5  #ifndef LIST_H
6  #define LIST_H
7
8  #include <iostream>
9
10 // Pr-dclarations ncessaires pour l'utilisation des classes amies
11 template <class T> class List;
12 template <class T> class ListIterator;
13
14 // ========================================================================
15 // Classe: dfinition de la classe ListElement<T> dcrivant un lment de
16 //         liste.
17 //         Tous les membres de la classe ListElement sont dfinis privs,
18 //         ainsi seules les classes amies List et ListIterator y ont accs.
19 // ========================================================================
20 template <class T> class ListElement {
21   private:
22     // Valeur de l'lment
23     T _value;
24
25     // Membres pour le chainage
26     ListElement<T>* _prev;
27     ListElement<T>* _next;
28
29     // Constructeurs et destructeur : ils mettent  jour le chainage au sein de la
       liste
30     ListElement(const T& v) : _value(v), _prev(0), _next(0) {}
31     ListElement(ListElement<T>* p, ListElement<T>* n)
32       :  _prev(p), _next(n) {
```

```
33        if(n != 0) {
34          n->_prev = this;
35        }
36        if(p != 0) {
37          p->_next = this;
38        }
39      }
40      ListElement(const T& v,ListElement<T>* p, ListElement<T>* n)
41        : _value(v), _prev(p), _next(n) {
42        if(n != 0) {
43          n->_prev = this;
44        }
45        if(p != 0) {
46          p->_next = this;
47        }
48      }
49      ~ListElement() {
50        if(_prev != 0) {
51          _prev->_next = _next;
52        }
53        if(_next != 0) {
54          _next->_prev = _prev;
55        }
56      }
57
58      // Classes amies
59      friend class List<T>;
60      friend class ListIterator<T>;
61  };
62
63
64
65  // Prdclaration de List pour pouvoir pr-dclarer des oprateurs
66  template <class T> class List;
67
68  // Pr-dclaration pouvoir dclarer ces oprateurs amis de List
69  template <class T> std::ostream& operator<<(std::ostream& out, const List<T>&
        lref);
70  template <class T> std::istream& operator>>(std::istream& in, List<T>& lref);
71
72  // ====================================================================
73  // Classe: dfinition de la classe List<T> dcrivant des listes gnriques
74  //         La gestion de la liste est en double chainage avec deux lments
75  //         fictifs _head et _tail pour grer le chainage.
76  // ====================================================================
77  template <class T> class List {
78    private:
79      // Les deux lments fictifs
80      ListElement<T>* _head;
```

```
81      ListElement<T>* _tail;
82
83      // Nombre d'lments contenus
84      int _card;
85
86      // ========================================================================
87      // But: suppression des lments contenus (mais pas les lments fictifs)
88      // ========================================================================
89      void _freelist() {
90        ListElement<T>* tmp = _head->_next;
91        while(tmp != _tail) {
92          ListElement<T>* n = tmp->_next;
93          _card--;
94          delete tmp;
95          tmp = n;
96        }
97      }
98
99    protected:
100     // Dfinition du type numr eListPosition
101     // Dfinition des diffrents emplacements d'insertion d'un nouvel lment
102     // dans une liste.
103     enum eListPosition {
104       LP_first = -2, // en dbut de liste
105       LP_last = -1,  // en fin de liste
106       LP_pos = 0     //  une position donne (entre 1 et le cardinal de la liste)
107     };
108
109   public:
110     // ========================================================================
111     // But: constructeur par dfaut : cration d'une liste vide
112     // ========================================================================
113     List(): _card(0) {
114       _head = new ListElement<T>(0, 0);
115       _tail = new ListElement<T>(_head, 0);
116       _head->_next = _tail;
117     }
118
119     // ========================================================================
120     // But: constructeur par recopie
121     // ========================================================================
122     List(const List<T>& lref): _card(0) {
123       _head = new ListElement<T>(0, 0);
124       _tail = new ListElement<T>(_head, 0);
125       _head->_next=_tail;
126
127       ListElement<T>* tmp = lref._head->_next;
128       while(tmp != lref._tail) {
129         ListElement<T>* toadd;
```

```
130         // Ajout  la fin de la liste
131         toadd = new ListElement<T>(tmp->_value,_tail->_prev,_tail);
132         _card++;
133         tmp = tmp->_next;
134       }
135     }
136 
137     // ========================================================================
138     // But: destructeur
139     // ========================================================================
140     ~List() {
141       _freelist();
142       delete _head;
143       delete _tail;
144     }
145 
146     // ========================================================================
147     // But: oprateur d'affectation
148     // ========================================================================
149     List<T>& operator=(const List<T>& lref) {
150       if(this != &lref) {
151         _freelist();
152         ListElement<T>* tmp = lref._head->_next;
153         while(tmp != lref._tail) {
154           ListElement<T>* toadd;
155           // Ajout  la fin de la liste
156           toadd = new ListElement<T>(tmp->_value,_tail->_prev,_tail);
157           _card++;
158           tmp = tmp->_next;
159         }
160       }
161       return *this;
162     }
163 
164     // ========================================================================
165     // But: test d'appartenance d'un lment  une liste, rend l'index si prsent
166     // ========================================================================
167     bool operator==(const T& v) const {
168       bool present = false;
169       ListElement<T>* tmp = _head->_next;
170       while(tmp!=_tail && !present) {
171         if(tmp->_value == v) {
172           present = true;
173         }
174         tmp = tmp->_next;
175       }
176       return present;
177     }
178 
```

```
179     // ========================================================================
180     // But: cardinal de la liste
181     // ========================================================================
182     int card() const {
183       return _card;
184     }
185 
186     // ========================================================================
187     // But: ajout d'un lment  une liste (par dfaut en tte de liste)
188     // ========================================================================
189     virtual void addElement(const T& v, eListPosition pos = LP_first) {
190       ListElement<T>* toadd;
191       switch (pos) {
192         case LP_first:  // Ajout en dbut
193           toadd = new ListElement<T>(v, _head, _head->_next);
194           _card++;
195           break;
196         case LP_last: // Ajout en fin
197           toadd = new ListElement<T>(v, _tail->_prev, _tail);
198           _card++;
199           break;
200         case LP_pos: // Pas d'ajout en position 0 de la liste
201         default:
202           int realpos = pos;
203           ListElement<T>* tmp = _head->_next;
204           while(tmp!=_tail && realpos>1) {
205             tmp = tmp->_next;
206             realpos--;
207           }
208           // Ajout effectif de l'lment
209           if(realpos == 0) {
210             toadd = new ListElement<T>(v, tmp->_prev,tmp);
211           }
212           _card++;
213           break;
214       }
215     }
216 
217     // ========================================================================
218     // But: ajout d'une liste  une autre liste (par dfaut en tte de liste)
219     // ========================================================================
220     void addAllElements(const List<T>& list, eListPosition pos = LP_first) {
221       ListIterator<T> it = list.beg();
222       while(!it.finished()) {
223         this->addElement(it.get(), pos);
224         it++;
225       }
226     }
227 
```

```
228     // ========================================================================
229     // But: oprateur d'ajout d'un lment  une liste (l'ajout se fait en
230     //    dbut de liste)
231     // ========================================================================
232     List<T> operator+(const T& v) const {
233       List<T> lres(*this);
234       lres.addElement(v, LP_first);
235       return lres;
236     }
237 
238     // ========================================================================
239     // But: oprateur de suppression d'un lment d'une liste
240     // ========================================================================
241     List<T> operator-(const T& v) const {
242       List<T> lres(*this);
243       lres.delElement(v);
244       return lres;
245     }
246 
247     // ========================================================================
248     // But: suppression d'un lment (le premier trouve uniquement)
249     // ========================================================================
250     void delElement(const T& v) {
251       // Recherche de l'lment
252       ListElement<T>* tmp = _head->_next;
253       while(tmp!=_tail && tmp->_value!=v) {
254         tmp = tmp->_next;
255       }
256       // Si l'lment a t trouv, le dtruire
257       if(tmp != _tail) {
258         delete tmp;
259         _card--;
260       }
261     }
262 
263     // ========================================================================
264     // But: supprime tous les elements d'une liste  la liste courante.
265     // ========================================================================
266     void delAllElements(const List<T>& list) {
267       ListIterator<T> it = list.beg();
268       while(!it.finished()) {
269         this->delElement(it.get());
270         it++;
271       }
272     }
273 
274     // ========================================================================
275     // But: accs  un lment donn de la liste en donnant un indice
276     // ========================================================================
```

```
277     T& operator[](const int& idx) const {
278       int id = 1;
279       ListElement<T>* tmp = _head->_next;
280       while(tmp!=_tail && id!=idx) {
281         tmp = tmp->_next;
282         id++;
283       }
284       return tmp->_value;
285     }
286 
287     // ========================================================================
288     // But: itrateur de liste  partir du dbut
289     // ========================================================================
290     ListIterator<T> beg() const {
291       return ListIterator<T>(*this);
292     }
293 
294     // ========================================================================
295     // But: itrateur de liste  partir de la fin
296     // ========================================================================
297     ListIterator<T> end() {
298       ListIterator<T> res(*this);
299       res._crtelt = _tail->_prev;
300       return res;
301     }
302 
303     // ========================================================================
304     // But: itrateur de liste  partir d'une position donne
305     // ========================================================================
306     ListIterator<T> pos(const int& idx) {
307       int realpos(idx);
308       ListIterator<T> res(*this);
309       ListElement<T>* tmp = _head->_next;
310       while(tmp!=_tail && realpos>1) {
311         tmp = tmp->_next;
312         realpos--;
313       }
314       res._crtelt = tmp;
315       return res;
316     }
317 
318     // ========================================================================
319     // But: oprateur d'affichage d'une liste dans un flux
320     // ========================================================================
321     friend std::ostream& operator<< <T>(std::ostream& out,const List<T>& lref);
322 
323     // ========================================================================
324     // But: oprateur de lecture d'une liste dans un flux
325     // ========================================================================
```

```
326     friend std::istream& operator>> <T>(std::istream& in,List<T>& lref);
327 
328     // Classe amie
329     friend class ListIterator<T>;
330 };
331 
332 
333 
334 // ========================================================================
335 // Classe: dfinition de la classe ListIterator<T> de parcours des listes
336 // ========================================================================
337 template <class T> class ListIterator {
338     // La liste de rfrence
339     const List<T>& _listref;
340     // La position courante
341     ListElement<T>* _crtelt;
342 
343     // Constructeur : on ne cre un itrateur que grce  la classe List
344     ListIterator(const List<T>& lref): _listref(lref) {
345       _crtelt = lref._head->_next;
346     }
347 
348   public:
349     // Destructeur */
350     ~ListIterator() {}
351     // Fin du parcours (on est sur l'un des lments fictifs
352     int finished() const {
353       return _crtelt==_listref._tail || _crtelt==_listref._head;
354     }
355     // Parcours en marche avant
356     ListIterator<T>& operator++() {
357       _crtelt = _crtelt->_next; return *this;
358     }
359     // Parcours en marche arrire
360     ListIterator<T>& operator--() {
361       _crtelt = _crtelt->_prev; return *this;
362     }
363     // Rcupration de l'lment courant
364     T& get() {
365       return _crtelt->_value;
366     }
367 
368     // Classe amie
369     friend class List<T>;
370 };
371 
372 template <class T>
373 std::ostream& operator<<(std::ostream& out, const List<T>& lref) {
374   out << lref.card() << "    ";
```

```
375   for(ListIterator<T> iterlst = lref.beg(); !(iterlst.finished()); ++iterlst) {
376     out << iterlst.get() << "  ";
377   }
378   return out;
379 }
380 
381 template <class T>
382 std::istream& operator>>(std::istream& in, List<T>& lref) {
383   int nb;
384   in >> nb;
385   for(int i = 0; i < nb; i++) {
386     T tmp;
387     in >> tmp;
388     lref.addElement(tmp, List<T>::LP_last);
389   }
390   return in;
391 }
392 
393 #endif
```