```
/*
 * File:   Lista_TCP.h
 * Author: inf-coccodi-de
 *
 * Created on 12 febbraio 2009, 8.15
 */

#ifndef _LISTA_TCP_H
#define _LISTA_TCP_H

#include <stdio.h>
#include "addLib.h"
#define MAX_ETH 1500

/* ------------------------------------------------------------------------ */

class Node
{
private:
    Node* next;
    Node* previous;
public:
    Node(Node*,Node*);
    Node();
    virtual ~Node();
    virtual void show();
    void set_key();
    void set_next(Node*);
    void set_previous(Node*);
    int get_key();
    Node* get_previous();
    Node* get_next();
};

/* ------------------------------------------------------------------------ */

Node::Node(Node* newPrevious, Node* newNext)
{
    next = newNext;
    previous=newPrevious;
}

Node::Node()
{
    next = NULL;
    previous=NULL;
}

Node::~Node()
{
}

void Node::show()
{
```

```
55      printf("precedente:%p__successivo:%p\n",previous ,next);
56      printf("puntatore istanza corrente %p\n", this);
57  }
58
59  void Node::set_next(Node * n_nodo)
60  {
61      next = n_nodo;
62  }
63
64  void Node::set_previous(Node* n_nodo)
65  {
66      previous = n_nodo;
67  }
68
69  Node* Node::get_next()
70  {
71      return (next);
72  }
73
74  Node* Node::get_previous()
75  {
76      return (previous);
77  }
78
79  /* ------------------------------------------------------------------------
      */
80
81  class Connection:public Node
82  {
83  private:
84      Address* myAddr;
85      int conn_id;
86  public:
87      Connection(Address*,Node*,Node*,int);
88      Connection(int);
89      Connection(Address*,int);
90      Connection(char*,int,int);
91      ~Connection();
92      void show();
93      void setAddr(Address*);
94      Address* getAddr();
95      void setConn_id(int);
96      int getConn_id();
97      void invia(char*);
98      char* ricevi();
99
100
101 };
102
103 /* ------------------------------------------------------------------------
      */
104
105 Connection::Connection(Address* _a,Node* newPrevious, Node* newNext,int
      _conn_id)
106 :Node(newPrevious,newNext)
107 {
```

```
108   myAddr=_a;
109   conn_id=_conn_id;
110  }
111
112  Connection::Connection(Address* _a,int _conn_id)
113  :Node()
114  {
115      myAddr=_a;
116      conn_id=_conn_id;
117  }
118
119  Connection::Connection(char* _c,int _i,int _conn_id)
120  :Node()
121  {
122      myAddr=new Address(_i,_c);
123      conn_id=_conn_id;
124  }
125
126  Connection::Connection(int _conn_id):
127  Node()
128  {
129      myAddr=new Address(8000,"0.0.0.0");
130      conn_id=_conn_id;
131  }
132
133  Connection::~Connection()
134  {
135      //Node::~Node();
136  /* DEBUG*/ printf ("---->distruttore Connection\n");
137
138      shutdown(conn_id,SHUT_RDWR);
139      delete(myAddr);
140  }
141
142  void Connection::show()
143  {
144      Node::show();
145      myAddr->stampaAdd();
146      printf("\nConn_id: %d\n",conn_id);
147  }
148
149  void Connection::setAddr(Address* _Add)
150  {
151      myAddr = _Add;
152  }
153
154  Address* Connection::getAddr()
155  {
156      return (myAddr);
157  }
158
159  void Connection::setConn_id(int Replacement)
160  {
161      conn_id = Replacement;
162  }
163
```

```
164 int Connection::getConn_id()
165 {
166     return (conn_id);
167 }
168
169 void Connection::invia(char* msg)
170 {
171  int len_tx;
172  int lenMsg;
173  lenMsg=lenStr(msg); //inserisci in lenMsg la lunghezza del messaggio
174  len_tx=send(conn_id,msg,lenMsg,0);//funzione che invia una stringa
175  if(len_tx!=lenStr(msg)) //se la lunghezza del messaggio inviato
       effettivamente e' minore della lunghezza del messaggio originale
176  {
177      errore("send()=",len_tx);//gestione dell'errore in invio
178  }
179 };
180
181 char* Connection::ricevi()
182 {
183     int rx_len;
184     char* buffer;//variabile che conterra' il messaggio ricevuto
185     buffer=(char*)malloc(sizeof(char)*1501);
186     rx_len=recv(conn_id,buffer,MAX_ETH,0);
187      if(rx_len>0)//se il messaggio ricevuto ha almeno un carattere al'interno
188     {
189           fflush(stdout);
190         *(buffer+(rx_len))='\0';//inserisci il valore di fine-stringa ad essa
191           // printf("\n----->Interna: %s<------\n",buffer);
192           //fflush(stdout);
193     }
194     else //altrimenti
195     {
196         //errore("recv()=",rx_len);//ritorna il genere di errore riscontrato
197         return (NULL);
198     }
199     return (buffer); //ritorna la copia del messaggio ricevuto
200 }
201
202 /* ------------------------------------------------------------------------
      */
203
204 class List; //prototipo di classe
205
206 /* ------------------------------------------------------------------------
      */
207
208 class Iterator
209 {
210 private:
211     List* myList;
212     Node* current;
213 public:
214     Iterator(List*,Node*);
215     Iterator(List*);
216     Iterator();
```

```
217     ~Iterator();
218     Node* getCurrent();
219     void setCurrent(Node*);
220     void moveFirst();
221     void moveLast();
222     void movePrevious();
223     void moveNext();
224     int isFirst();
225     int isLast();
226     void showCurrent();
227 };
228
229 /* ------------------------------------------------------------------------ */
230
231 class List
232 {
233 private:
234     Node* first;
235     Node* last;
236     void deleteList(Node*);
237     void showList(Node*);
238 public:
239     List();
240     ~List();
241     int is_empty();
242     void show();
243     Node* getFirst();
244     Node* getLast();
245     void addOnHead(Node*);
246     void addOnTail(Node*);
247     void removeFromHead();
248     void removeFromTail();
249     Iterator* createIterator();
250     bool remove(Node*);
251     bool exists(Node*);
252 };
253
254 /* ------------------------------------------------------------------------ */
255
256 List::List()
257 {
258     first = NULL;
259     last=NULL;
260 }
261
262 List::~List()
263 {
264     deleteList(first);
265     first=NULL;
266     last=NULL;
267 }
268
269 int List::is_empty()
270 {
```

```
271     return ((first) ? 0 : 1);
272 }
273
274 void List::show()
275 {
276     showList(first);
277 }
278
279 void List::deleteList(Node* a)
280 {
281     if (a!=NULL)
282     {
283         deleteList(a->get_next());
284         delete(a);
285     }
286 }
287
288 void List::showList(Node* a)
289 {
290     if (a)
291     {
292         a->show();
293         showList(a->get_next());
294     }
295 }
296
297 void List::addOnHead(Node* incoming)
298 {
299     if(last==NULL)
300     {
301             incoming->set_next(NULL);
302             incoming->set_previous(NULL);
303             last=incoming;
304             first=incoming;
305     }
306     else
307     {
308             first->set_previous(incoming);
309             incoming->set_next(first);
310             incoming->set_previous(NULL);
311             first=incoming;
312     }
313 }
314
315 void List::addOnTail(Node* incoming)
316 {
317     if(first==NULL)
318     {
319             incoming->set_next(NULL);
320             incoming->set_previous(NULL);
321             last=incoming;
322             first=incoming;
323     }
324     else
325     {
326             last->set_next(incoming);
```

```
327             incoming->set_previous(last);
328             incoming->set_next(NULL);
329             last=incoming;
330     }
331 }
332
333 Node* List::getFirst()
334 {
335     return (first);
336 }
337
338 Node* List::getLast()
339 {
340     return (last);
341 }
342
343 void List::removeFromHead()
344 {
345     Node* nodo;
346     nodo=first;
347     if(first!=NULL)
348     {
349         if(nodo->get_next()==NULL)
350         {
351             delete(nodo);
352             first=NULL;
353             last=NULL;
354         }
355         else
356         {
357             nodo=nodo->get_next();
358             delete(first);
359             first=nodo;
360             first->set_previous(NULL);
361         }
362     }
363 }
364
365 void List::removeFromTail()
366 {
367     Node* nodo;
368     nodo=last;
369     if(last!=NULL)
370     {
371         if(nodo->get_previous()==NULL)
372         {
373             delete(nodo);
374             first=NULL;
375             last=NULL;
376         }
377         else
378         {
379             nodo=nodo->get_previous();
380             delete(last);
381             last=nodo;
382             last->set_next(NULL);
```

```
383         }
384     }
385 }
386
387  Iterator* List::createIterator()
388  {
389    return (new Iterator(this));
390  }
391
392 bool List::remove(Node* nodo)
393 {
394  Node   *curry,*next,*prev;
395
396  for (curry=first;(curry);curry=curry->get_next())
397    if (curry==nodo)
398        {
399     next = nodo->get_next();
400     prev = nodo->get_previous();
401     if (next)
402       next->set_previous(prev);
403     else
404       last=prev;
405     if(prev)
406       prev->set_next(next);
407     else
408       first=next;
409     delete(nodo);
410     return true;
411        }
412        return false;
413 }
414
415 bool List::exists(Node* nodo)
416 {
417  Node* curr;
418
419  for(curr=first;curr;curr=curr->get_next())
420     if(curr==nodo)
421       return true;
422  return false;
423 }
424  /* ------------------------------------------------------------------------ ↵
      */
425
426 Iterator::Iterator(List* _l, Node* _n) {
427     myList = _l;
428     current = _n;
429 }
430
431 Iterator::Iterator(List* _l) {
432     myList = _l;
433     current = _l->getFirst();
434 }
435
436 Iterator::Iterator() {
437     myList = new List();
```

```
438     current = NULL;
439 }
440
441 Iterator::~Iterator() {
442
443 }
444
445 Node* Iterator::getCurrent() {
446     return (current);
447 }
448
449 void Iterator::setCurrent(Node* _n) {
450     current = _n;
451 }
452
453 void Iterator::moveFirst() {
454     current = myList->getFirst();
455 };
456
457 void Iterator::moveLast() {
458     current = myList->getLast();
459 }
460
461 void Iterator::movePrevious() {
462     if ((myList->getLast() != NULL) && (current->get_previous() != NULL)) {
463         current = current->get_previous();
464     }
465 }
466
467 void Iterator::moveNext() {
468     if ((myList->getLast() != NULL) && (current->get_next() != NULL)) {
469         current = current->get_next();
470     }
471 }
472
473 int Iterator::isFirst() {
474     return ((current->get_previous()) ? 0 : 1);
475 }
476
477 int Iterator::isLast() {
478     return ((current->get_next()) ? 0 : 1);
479 }
480
481 void Iterator::showCurrent() {
482     current->show();
483 }
484
485 #endif  /* _LISTA_TCP_H */
486
487
```