```
#ifndef LINKEDLIST_CLASS
#define LINKEDLIST_CLASS

#include <iostream.h>
#include <stdlib.h>

#ifndef NULL
const int NULL = 0;
#endif  // NULL

#include "node.h"

template <class T>
class SeqListIterator;

template <class T>
class LinkedList
{
   private:
      // pointers maintain access to front and rear of list
      Node<T> *front, *rear;

      // used for data retrieval, insertion and deletion
      Node<T> *prevPtr, *currPtr;

      // number of elements in the list
      int size;

      // position in list. used by Reset method
      int position;
      // private methods to allocate and deallocate nodes
      Node<T> *GetNode(const T& item,Node<T> *ptrNext=NULL);
      void FreeNode(Node<T> *p);

      // copies list L to current list
      void CopyList(const LinkedList<T>& L);

   public:
      // constructors
      LinkedList(void);
      LinkedList(const LinkedList<T>& L);

      // destructor
      ~LinkedList(void);

      // assignment operator
      LinkedList<T>& operator= (const LinkedList<T>& L);

      // methods to check list status
      int ListSize(void) const;
      int ListEmpty(void) const;

      // Traversal methods
      void Reset(int pos = 0);
      void Next(void);
      int EndOfList(void) const;
      int CurrentPosition(void) const;

      // Insertion methods
      void InsertFront(const T& item);
      void InsertRear(const T& item);
      void InsertAt(const T& item);
      void InsertAfter(const T& item);

      // Deletion methods
      T DeleteFront(void);
      void DeleteAt(void);

      // Data retrieval/modification
      T& Data(void);

      // method to clear the list
      void ClearList(void);

      // this class (Ch. 12) needs access to front
      friend class SeqListIterator<T>;
};

template <class T>
Node<T> *LinkedList<T>::GetNode(const T& item,
                      Node<T>* ptrNext)
{
   Node<T> *p;

   p = new Node<T>(item,ptrNext);
   if (p == NULL)
   {
      cout << "Memory allocation failure!\n";
      exit(1);
   }
   return p;
}

template <class T>
void LinkedList<T>::FreeNode(Node<T> *p)
{
   delete p;
}

// copy L to the current list, which is assumed to be empty
template <class T>
void LinkedList<T>::CopyList(const LinkedList<T>& L)
{
   // use p to chain through L
   Node<T> *p = L.front;
   int pos;

   // insert each element in L at the rear of current object
   while (p != NULL)
   {
      InsertRear(p->data);
      p = p->NextNode();
```

```
   }

   // if list is empty return
   if (position == -1)
      return;

   // reset prevPtr and currPtr in the new list
   prevPtr = NULL;
   currPtr = front;
   for (pos = 0; pos != position; pos++)
   {
      prevPtr = currPtr;
      currPtr = currPtr->NextNode();
   }
}

// create empty list by setting pointers to NULL, size to 0
// and list position to -1
template <class T>
LinkedList<T>::LinkedList(void): front(NULL), rear(NULL),
      prevPtr(NULL),currPtr(NULL), size(0), position(-1)
{}

template <class T>
LinkedList<T>::LinkedList(const LinkedList<T>& L)
{
   front = rear = NULL;
   prevPtr = currPtr = NULL;
   size = 0;
   position = -1;
   CopyList(L);
}

template <class T>
void LinkedList<T>::ClearList(void)
{
   Node<T> *currPosition, *nextPosition;

   currPosition = front;
   while(currPosition != NULL)
   {
      // get address of next node and delete current node
      nextPosition = currPosition->NextNode();
      FreeNode(currPosition);
      currPosition = nextPosition;  // Move to next node
   }
   front = rear = NULL;
   prevPtr = currPtr = NULL;
   size = 0;
   position = -1;
}

template <class T>
LinkedList<T>::~LinkedList(void)
{
   ClearList();
```

```
}

template <class T>
LinkedList<T>& LinkedList<T>::operator=
               (const LinkedList<T>& L)
{
   if (this == &L)      // Can't assign list to itself
      return *this;

   ClearList();
   CopyList(L);
   return *this;
}

template <class T>
int LinkedList<T>::ListSize(void) const
{
   return size;
}

template <class T>
int LinkedList<T>::ListEmpty(void) const
{
   return size == 0;
}

// move prevPtr and currPtr forward one node
template <class T>
void LinkedList<T>::Next(void)
{
   // if traversal has reached the end of the list or
   // the list is empty, just return
   if (currPtr != NULL)
   {
      // advance the two pointers one node forward
      prevPtr = currPtr;
      currPtr = currPtr->NextNode();
      position++;
   }
}

// True if the client has traversed the list
template <class T>
int LinkedList<T>::EndOfList(void) const
{
   return currPtr == NULL;
}

// return the position of the current node
template <class T>
int LinkedList<T>::CurrentPosition(void) const
{
   return position;
}

// reset the list position to pos
```

```
template <class T>
void LinkedList<T>::Reset(int pos)
{
   int startPos;

   // if the list is empty, return
   if (front == NULL)
      return;

   // if the position is invalid, terminate the program
   if (pos < 0 || pos > size-1)
   {
      cerr << "Reset: Invalid list position: " << pos
           << endl;
      return;
   }

   // move list traversal mechanism to node pos
   if(pos == 0)
   {
      // reset to front of the list
      prevPtr = NULL;
      currPtr = front;
      position = 0;
   }
   else
   // reset currPtr, prevPtr, and position
   {
       currPtr = front->NextNode();
       prevPtr = front;
       startPos = 1;
         // move right until position == pos
         for(position=startPos; position != pos; position++)
         {
             // move both traversal pointers forward
             prevPtr = currPtr;
             currPtr = currPtr->NextNode();
      }
   }
}

// return a reference to the data value in the current node
template <class T>
T& LinkedList<T>::Data(void)
{
   // error if list is empty or traversal completed
   if (size == 0 || currPtr == NULL)
   {
      cerr << "Data: invalid reference!" << endl;
      exit(1);
   }
   return currPtr->data;
}

// Insert item at front of list
template <class T>
void LinkedList<T>::InsertFront(const T& item)
{
   // call Reset if the list is not empty
   if (front != NULL)
      Reset();
   InsertAt(item);        // inserts at front
}

// Insert item at rear of list
template <class T>
void LinkedList<T>::InsertRear(const T& item)
{
   Node<T> *newNode;

   prevPtr = rear;
   newNode = GetNode(item);// create the new node
   if (rear == NULL)                  // if list empty, insert
at front
      front = rear = newNode;
   else
   {
      rear->InsertAfter(newNode);
      rear = newNode;
   }
   currPtr = rear;
   position = size;
   size++;
}

// Insert item at the current list position
template <class T>
void LinkedList<T>::InsertAt(const T& item)
{
   Node<T> *newNode;

   // two cases: inserting at the front or inside the list
   if (prevPtr == NULL)
   {
      // inserting at the front of the list. also places
      // node into an empty list
      newNode = GetNode(item,front);
      front = newNode;
   }
   else
   {
      // inserting inside the list. place node after prevPtr
      newNode = GetNode(item);
      prevPtr->InsertAfter(newNode);
   }

   // if prevPtr == rear, we are inserting into empty list
   // or at rear of non-empty list; update rear and position
   if (prevPtr == rear)
   {
      rear = newNode;
      position = size;
```

```
   }

   // update currPtr and increment the list size
   currPtr = newNode;
   size++;              // increment list size
}

// Insert item after the current list position
template <class T>
void LinkedList<T>::InsertAfter(const T& item)
{
   Node<T> *p;

   p = GetNode(item);
   if (front == NULL)       // inserting into an empty list
   {
      front = currPtr = rear = p;
      position = 0;
   }
   else
   {
      // inserting after last node of list
      if (currPtr == NULL)
        currPtr = prevPtr;
      currPtr->InsertAfter(p);
      if (currPtr == rear)
      {
        rear = p;
        position = size;
      }
      else
           position++;
      prevPtr = currPtr;
      currPtr = p;
   }
   size++;              // increment list size
}

// Delete the node at the front of list
template <class T>
T LinkedList<T>::DeleteFront(void)
{
   T item;

   Reset();
   if (front == NULL)
   {
      cerr << "Invalid deletion!" << endl;
      exit(1);
   }
   item = currPtr->data;
   DeleteAt();
   return item;
}

// Delete the node at the current list position
```

```
template <class T>
void LinkedList<T>::DeleteAt(void)
{
   Node<T> *p;

   // error if empty list or at end of list
   if (currPtr == NULL)
   {
      cerr << "Invalid deletion!" << endl;
      exit(1);
   }

   // deletion must occur at front node or inside the list
   if (prevPtr == NULL)
   {
      // save address of front and unlink it. if this
      // is the last node, front becomes NULL
      p = front;
      front = front->NextNode();
   }
   else
      // unlink interior node after prevPtr. save address
      p = prevPtr->DeleteAfter();

   // if rear is deleted, new rear is prevPtr and position
   // is decremented; otherwise, position is the same
   // if p was last node, rear = NULL and position = -1
   if (p == rear)
   {
      rear = prevPtr;
      position--;
   }

   // move currPtr past deleted node. if p is last node
   // in the list, currPtr becomes NULL
   currPtr = p->NextNode();

   // free the node and decrement the list size
   FreeNode(p);
   size--;
}

#endif  // LINKEDLIST_CLASS
```

```
template <class T>
class Node
{
    private:
           // next is the address of the following node
        Node<T> *next;
    public:
      // the data is public
        T data;

           // constructor
        Node (const T& item, Node<T>* ptrnext = NULL);

        // list modification methods
        void InsertAfter(Node<T> *p);
        Node<T> *DeleteAfter(void);

        // obtain the address of the next node
        Node<T> *NextNode(void) const;
};
// constructor. initialize data and pointer members
template <class T>
Node<T>::Node(const T& item, Node<T>* ptrnext) :
            data(item), next(ptrnext)
{}
// return value of private member next
template <class T>
Node<T> *Node<T>::NextNode(void) const
{
    return next;
}
// insert a node p after the current one
template <class T>
void Node<T>::InsertAfter(Node<T> *p)
{
      // p points to successor of the current node,
      // and current node  points to p.
    p->next = next;
    next = p;
}
// delete the node following current and return its address
template <class T>
Node<T> *Node<T>::DeleteAfter(void)
{
      // save address of node to be deleted
    Node<T> *tempPtr = next;

      // if there isn't a successor, return NULL
    if (next == NULL)
        return NULL;
      // current node points to successor of tempPtr.
    next = tempPtr->next;

      // return the pointer to the unlinked node
    return tempPtr;
}
```