```
/*****************************************************
 * Jared Dembrun
 * 10/11/2013
 * List.cc
 *
 * An ADT for storing integer data and dynamic in memory allocation
 *****************************************************/

#include <iostream>
using namespace std;

const int MAX_SIZE = 30;

class List
{
        private:
                // list node definition
                struct Node
                {
                        int data;
                        Node *link;
                };

                Node *head;     // the head of the list
                Node *tail;     // the tail of the list
                Node *curr;     // the current position in the list
                int num_items;  // the number of items in the list

        public:
                // constructor
                // remember that an empty list has a "size" of -1 and its "position" is at -1
                List()
                {
                        head = NULL;
                        tail = NULL;
                        curr = NULL;
                        num_items = 0;
                }

                // copy constructor
                // clones the list l and sets the last element as the current
                List(const List& l)
                {
                        (*this).head = NULL;
                        (*this).curr = head;
                        (*this).tail = head;
                        (*this).num_items = 0;
                        (*this) = l;
                }

                // copy constructor
                // clones the list l and sets the last element as the current
                void operator=(const List& l)
                {
                        Node *iter = l.head;
                        for(int i = 0; i < l.num_items; i++)
                        {
                                (*this).InsertBefore((*iter).data);
                                iter = (*iter).link;
                        }

                }

                // navigates to the beginning of the list
                void First()
                {
                        curr = head;
```

```
	}

	// navigates to the end of the list
	// the end of the list is at the last valid item in the list
	void Last()
	{
		curr = tail;
	}

	// navigates to the specified element (0-index)
	// this should not be possible for an empty list
	// this should not be possible for invalid positions
	void SetPos(int pos)
	{
		if(GetSize() > pos)//num_items is non-zero indexed and pos is zero-indexed
		{
			curr = head;
			for(int i = 0; i <= pos; i++)
				curr = ((*curr).link);
			cout << "got here" << endl;
		}
	}

	// navigates to the previous element
	// this should not be possible for an empty list
	// there should be no wrap-around
	void Prev()
	{
		if(curr != head)
		{
			Node *iter, *temp;
			iter = head;
			for(int i = 0; i < num_items; i++)
			{
				temp = iter;//save the current iteration
				iter = (*iter).link;//progress the iterator by one node
				if(iter == curr)
				{
					curr = temp;//once we have found curr, set it back one
node in the list
					break;
				}
			}
		}
	}

	// navigates to the next element
	// this should not be possible for an empty list
	// there should be no wrap-around
	void Next()
	{
		if((*curr).link != NULL)
			curr = (*curr).link;
	}

	// returns the location of the current element (or -1)
	int GetPos()
	{
		if(IsEmpty())
			return -1;

		Node *iter;
		iter = head;
		int pos = 0;
		while(iter != curr)
		{
			iter = (*iter).link;
```

```
                pos++;
        }
        return pos;
}

// returns the value of the current element (or -1)
int GetValue()
{
        if(!IsEmpty())
                return (*curr).data;
        else
                return -1;
}

// returns the size of the list
// size does not imply capacity
int GetSize()
{
        return num_items;
}

// inserts an item before the current element
// the new element becomes the current
// this should not be possible for a full list
void InsertBefore(int data)
{
        if(!IsFull() && !IsEmpty() && curr != head)
        {
                Node *temp;
                temp = curr;
                Prev();
                Node *nu = new Node;
                (*curr).link = nu;
                (*nu).link = temp;
                (*nu).data = data;
                num_items++;
                curr = (*curr).link;
        }
        else if(IsEmpty())
        {
                head = new Node;

                (*head).link = NULL;
                (*head).data = data;
                tail = head;
                curr = head;
                num_items++;
        }
        else if(curr == head)
        {
                Node *temp = new Node;
                (*temp).link = head;
                (*temp).data = data;
                head = temp;
                Prev();
                num_items++;
        }

}

// inserts an item after the current element
// the new element becomes the current
// this should not be possible for a full list
void InsertAfter(int data)
{
        if(curr != tail)
        {
```

```
                Next();
                InsertBefore(data);
        }
        else//if the data is inserted at the end, move the tail
        {

                if(IsEmpty())
                {
                        tail = new Node;
                        head = tail;
                        curr = tail;
                        (*tail).data = data;
                        (*tail).link = NULL;
                        num_items++;
                }
                else
                {
                        (*tail).link = new Node;
                        tail->link->data = data;
                        tail = (*tail).link;
                        curr = tail;
                        num_items++;
                        (*tail).link = NULL;
                }
        }
}

// removes the current element (collapsing the list)
// this should not be possible for an empty list
void Remove()
{
        if(!IsEmpty())
        {
                if(curr == head)
                {
                        Node *temp = head;
                        head = (*head).link;
                        curr = head;
                        delete temp;
                        temp = NULL;
                        num_items--;
                }
                else if(curr == tail)
                {
                        Prev();
                        delete tail;
                        tail = curr;
                        (*tail).link = NULL;
                        num_items--;
                }
                else
                {
                        Node *temp = curr;
                        Prev();
                        (*curr).link = (*temp).link;
                        delete temp;
                        temp = NULL;
                        num_items--;
                }

                if(num_items == 0)
                {
                        head = NULL;
                        curr = head;
                        tail = head;
                }
        }
```

```
		}

		// replaces the value of the current element with the specified value
		// this should not be possible for an empty list
		void Replace(int data)
		{
			if(!IsEmpty())
				(*curr).data = data;
		}

		// returns if the list is empty
		bool IsEmpty() const
		{
			return (num_items < 1);
		}

		// returns if the list is full
		bool IsFull()
		{
			return (num_items == MAX_SIZE);
		}

		// returns the concatenation of two lists
		// l should not be modified
		// l should be concatenated to the end of *this
		// the returned list should not exceed MAX_SIZE elements
		// the last element of the new list is the current
		List operator+(const List& l) const
		{
			List *nu = new List;
			Node *iter = (*this).head;

			while((iter != NULL) && !(*nu).IsFull())//At end of list, iter will be set to
tail.link, which equals NULL
			{
				(*nu).InsertAfter((*iter).data);
				iter = (*iter).link;
			}

			iter = l.head;

			while((iter != NULL) && !(*nu).IsFull())
			{
				(*nu).InsertAfter((*iter).data);
				iter = (*iter).link;
			}

			return (*nu);

		}

		// returns if two lists are equal (by value)
		bool operator==(const List& l) const
		{
			bool equal = ((*this).num_items == l.num_items);
			if(!equal)
				return equal;

			Node *iter1 = (*this).head;
			Node *iter2 = l.head;

			while((iter1 != NULL) && (iter2 != NULL))
			{
				equal = ((*iter1).data == (*iter2).data);
				if(!equal)
					return equal;
				iter1 = (*iter1).link;
```

```
                        iter2 = (*iter2).link;
                }
                return equal;
        }

        // returns if two lists are not equal (by value)
        bool operator!=(const List& l) const
        {
                return !((*this) == l);
        }

        // returns a string representation of the entire list (e.g., 1 2 3 4 5)
        // the string "NULL" should be returned for an empty list
        friend ostream& operator<<(ostream& out, const List &l)
        {
                if(!l.IsEmpty())
                {
                        Node *iter = l.head;
                        for(int i = 0; i < l.num_items; i++)
                        {
                                out << (*iter).data << " ";
                                iter = (*iter).link;
                        }
                }
                else
                        out << "NULL";
                return out;
        }
};
```