```
#include "List.H"

List::List()
        : _head(NULL), _current(NULL)
{
}

List::~List()
{
        _destroyList(_head);
        delete _head;
        _current = _head = NULL;
}

void
List::_destroyList(ListNode * l)
{
        if (l == NULL)
                return;

        _destroyList(l->_next);
        delete l;

        return;
}
```

```
List::List(const List & rhs)
{

}

List &
List::operator =(const List &rhs)
{
        return *this;
}

bool
List::insert(int x) {

        if (isEmpty()) {
                _head = new ListNode(x, NULL);
                _current = _head;
                return true;
        }

        if (isPastEnd())
                return false;


        //  If we're inserting at the beginning of the list,
```

```
        //  there's no previous node.
        if (_current == _head) {
                _head = _current = new ListNode(x, _head);
                return true;
        } else {
                ListNode * previous = _findPrevious(_current)
;
                ListNode * tmp = new ListNode(x, _current);
                previous->_next = tmp;
                _current = tmp;
        }

        return true;
}


ListNode *
List::_findPrevious(ListNode * l)
{
        ListNode * previous;

        if (l == NULL)
                return NULL;

        for (previous = _head;
             previous != NULL;
```

```
                previous = previous->_next) {
                        if (previous->_next == l)
                                return previous;
        }

        return NULL;
}


void
List::remove()
{
        if (isEmpty() || isPastEnd())
                return;

        ListNode * previous = _findPrevious(_current);
        previous->_next = _current->_next;
        delete _current;
        _current = previous->_next;
}

void
List::next()
{
        if (!isPastEnd())
                _current = _current->_next;
```

```
        else
                _current = NULL;
}

void
List::previous()
{
        if (!isPastEnd())
                _current = _findPrevious(_current);
}

int
List::current()
{
        if (isPastEnd()) {
                cerr << "Uh oh" << endl;
                return -1;
        } else
                return _current->_data;
}

void
List::reset()
{
        _current = _head;
}
```

```
bool
List::isEmpty()
{
        return _head == NULL;
}

bool
List::isPastEnd()
{
        return _current == NULL;
}

ostream &
operator <<(ostream & os, List & l)
{
        ListNode * old_current = l._current;

        if (l.isEmpty())
                return os << "()";

        os << "(";

        bool first_pass=true;
        for (l.reset() ; !l.isPastEnd(); l.next()) {
                if (first_pass) {
```

```
                    os << l.current();
                    first_pass = false;
                } else
                    os << "," << l.current();
        }

        l._current = old_current;
        os << ")";

        return os;
}
```

```
#ifndef CMSC202_LIST_H
#define CMSC202_LIST_H

#include <iostream.h>

//  Forward declaration of List
class List;

class ListNode {
private:
        ListNode * _next;
        int        _data;

        ListNode(int x, ListNode * next) : _next(next), _data
(x) {}

        friend class List;
};

class List {
public:
        List();
        ~List();
        List(const List &);
        List & operator =(const List &);
```

```
        bool insert(int);
        void remove();
        bool isEmpty();

        bool isPastEnd();
        void next();
        void previous();
        int  current();
        void reset();

private:
        ListNode * _head;
        ListNode * _current;

        ListNode * _findPrevious(ListNode *);
        void       _destroyList(ListNode *);

        friend ostream & operator <<(ostream &, List &);
};

ostream & operator <<(ostream & os, List & l);

#endif
```

```
#include <iostream.h>
#include "List.H"

void
main()
{
        List l;

        l.insert(9);
        l.insert(7);
        l.insert(5);
        l.insert(3);
        l.insert(1);

        cout << l << endl;
        cout << "Current is now at: " << l.current() << endl;

        l.next();
        l.insert(2);

        l.next();
        l.next();
        l.insert(4);

        l.next();
        l.next();
```

```
        l.insert(6);

        l.next();
        l.next();
        l.insert(8);

        cout << l << endl;

        l.previous();
        l.previous();

        cout << "Current is now at: " << l.current() << endl;

        l.remove();
        cout << l << endl;
}
```