PROJECT 2. Linked Lists, Stacks, Queues, STL
ESE 344 Software Techniques for Engineers
SUNY at Stony Brook, Murali Subbarao , Weight: 10%

DRAFT (Subject to minor changes)

Reference: **http://users.cs.fiu.edu/~weiss/dsaa_c++4/code/**

1. You are given the Square class in Chapter 1 (Fig. 1.23, page 40) and the List Class in Chapter 3 (pages 94-102). The corresponding source code is included here (copy and paste this stuff into your source code file. See **http://users.cs.fiu.edu/~weiss/dsaa_c++4/code/**). Change the type of side data member to be int instead of double for the purpose of this project.
   Modify this doubly linked list implementation of LIST data structure in the text book (copied below) to implement it as a **singly linked list,** and **whenever appropriate, check for the validity of iterators.** Each node item in the list must contain only one pointer to the **next** item on the list (and no pointer to the previous item).
   Add the following methods:
   Operator-- : set itr to previous item (prefix)
   operator--(int) : : set itr to previous item (postfix)
   Make minor modifications as needed (e.g. add `operator==` to Square class). Use these class definitions, and test them with a `main()` function which does the operations listed below. The goal of this project is to become familiar with the implementation of linked Lists, Stacks, and Queues, similar to those in the Standard Template Library (STL). Therefore, you will be demonstrating the operations listed below on these data structures.

```
//copy constructor
    List( const List & rhs )

// assignment operator
    List & operator= ( const List & rhs )

// stack and queue operations
    void push_front( const Object & x )
    void push_back( const Object & x )
    void pop_front( )
    void pop_back( )

// clear
   void clear( )
```

Your program should do the following:

a. First, create lists L1, L2, L3, and L4, containing zero Square elements.
   Next, insert 20 Square objects into L1, in a loop, as follows:
   While(L1.size()<=20) {
   - Generate a random integers k in the range 1 to 100 with `((rand() % 100)+1 ))`.
   - Construct a Square object x with side value k (change the type of side data member to be int instead of double for the purpose of this project).
   - Execute: L1.push_front(x)
   - Execute: L2.push_back(x)
   - Check if the list L3 already contains a Square object of size k. If it does not contain a Square of size k, then insert x into L3 at a random position p where p=0 if L3 is empty; otherwise p is a random number in the range 0 to L3.size(), generated with `rand() % (L3.size()+1).` (Note position 0 is the first element, and position L3.size() is the position after the last element).
   - If(L3.size()==5) {
     L4 = L3; // assignment operator
     L4.pop_front(); L4.pop_back(); }
   - If(L1.size()==5) {
     - Copy construct a new list object L5 with initial value of L1.
     - Delete all odd sized Squares (i.e. side value is one of 1, 3, 5, …,99) from list L5. After deleting each odd-sized square, use the operator--() to access the previous node if it exists, and then print the size of the square in that node.

     - Print L5, and Clear L5.}
   - Print L1, L2, L3, L4.

   } End While
   Clear L1, L2, L3, and L4, and print them all.

Some notes from the text book:

(page 101) In examining the code, we can see a host of errors that can occur and for which no checks are provided. For instance, iterators passed to `erase and insert` can be uninitialized or for the wrong list! Iterators can have ++ or * applied to them when they are already at the endmarker or are uninitialized. An uninitialized iterator will have current pointing at nullptr, so that condition is easily tested. The endmarker's next pointer points at nullptr, so testing for ++ or * on an endmarker condition is also easy. However, in order to determine if an iterator passed to erase or insert is an iterator for the correct list, the iterator must store an additional data member representing a pointer to the List from which it was constructed.

The following is suggested on page 102:

```
1 protected:
2 const List<Object> *theList; //the list for which the iterator belongs
3 Node *current;
4
5 const_iterator( const List<Object> & lst, Node *p )
6 : theList{ &lst }, current{ p }
7 {
8 }
```

If the above method poses problems at later stages, you may instead try the following option:

```
protected:
     Node* head;  // head of the list for which the iterator belongs
     Node* current;
     const_iterator(Node* p, Node* h) : current{ p }, head { h }
     { }
```

Note: instead of "throw IteratorOutOfBoundsException{ };" you may just print an error message and exit, e.g.
{ "cerr<< "invalid iterator. Exiting program. \n"; exit(1);}

A(page 102) : all
calls to iterator and const_iterator constructors that formerly took one parameter now take two, as in the begin method for List:

```
const_iterator begin( ) const
{
const_iterator itr{ *this, head };   // const_iterator itr{ head, head };
return ++itr;
      }
```

Reference: **http://users.cs.fiu.edu/~weiss/dsaa_c++4/code/**
std::list - cppreference.com
Containers library - cppreference.com

// Code for random number generation

```
#include <iostream>
using namespace std;
#include <cstdlib>   // for rand(), srand()
#include <ctime>     // for time()
#include <assert.h>
#include <math.h>    // for sqrt()
```

```
int main() {

      srand((unsigned int) time(NULL));// seed rand() with system time

      for (int i = 0; i<100; i++) {

                  cout<< (rand() % 100) <<endl; // limit data to 0 to 99.

      }

      cout << "Done \n";
      return 0;
}

// List from M. A. Weiss

#ifndef LIST_H
#define LIST_H

#include <algorithm>
using namespace std;

template <typename Object>
class List
{
  private:
    // The basic doubly linked list node.
    // Nested inside of List, can be public
    // because the Node is itself private
    struct Node
    {
        Object  data;
        Node   *prev;
        Node   *next;

        Node( const Object & d = Object{ }, Node * p = nullptr, Node * n =
nullptr )
          : data{ d }, prev{ p }, next{ n } { }

        Node( Object && d, Node * p = nullptr, Node * n = nullptr )
          : data{ std::move( d ) }, prev{ p }, next{ n } { }
    };

  public:
    class const_iterator
    {
```

```
public:

  // Public constructor for const_iterator.
  const_iterator( ) : current{ nullptr }
    { }

  // Return the object stored at the current position.
  // For const_iterator, this is an accessor with a
  // const reference return type.
  const Object & operator* ( ) const
    { return retrieve( ); }

  const_iterator & operator++ ( )
  {
      current = current->next;
      return *this;
  }

  const_iterator operator++ ( int )
  {
      const_iterator old = *this;
      ++( *this );
      return old;
  }

  const_iterator & operator-- ( )
  {
      current = current->prev;
      return *this;
  }

  const_iterator operator-- ( int )
  {
      const_iterator old = *this;
      --( *this );
      return old;
  }

  bool operator== ( const const_iterator & rhs ) const
    { return current == rhs.current; }

  bool operator!= ( const const_iterator & rhs ) const
    { return !( *this == rhs ); }

protected:
  Node *current;
```

```
    // Protected helper in const_iterator that returns the object
    // stored at the current position. Can be called by all
    // three versions of operator* without any type conversions.
    Object & retrieve( ) const
      { return current->data; }

    // Protected constructor for const_iterator.
    // Expects a pointer that represents the current position.
    const_iterator( Node *p ) :  current{ p }
      { }

    friend class List<Object>;
};

class iterator : public const_iterator
{
  public:

    // Public constructor for iterator.
    // Calls the base-class constructor.
    // Must be provided because the private constructor
    // is written; otherwise zero-parameter constructor
    // would be disabled.
    iterator( )
      { }

    Object & operator* ( )
      { return const_iterator::retrieve( ); }

    // Return the object stored at the current position.
    // For iterator, there is an accessor with a
    // const reference return type and a mutator with
    // a reference return type. The accessor is shown first.
    const Object & operator* ( ) const
      { return const_iterator::operator*( ); }

    iterator & operator++ ( )
    {
        this->current = this->current->next;
        return *this;
    }

    iterator operator++ ( int )
    {
        iterator old = *this;
```

```
            ++( *this );
            return old;
        }

        iterator & operator-- ( )
        {
            this->current = this->current->prev;
            return *this;
        }

        iterator operator-- ( int )
        {
            iterator old = *this;
            --( *this );
            return old;
        }

      protected:
        // Protected constructor for iterator.
        // Expects the current position.
        iterator( Node *p ) : const_iterator{ p }
          { }

        friend class List<Object>;
    };

  public:
    List( )
      { init( ); }

    ~List( )
    {
        clear( );
        delete head;
        delete tail;
    }

    List( const List & rhs )
    {
        init( );
        for( auto & x : rhs )
            push_back( x );
    }

    List & operator= ( const List & rhs )
    {
```

```
    List copy = rhs;
    std::swap( *this, copy );
    return *this;
}


List( List && rhs )
  : theSize{ rhs.theSize }, head{ rhs.head }, tail{ rhs.tail }
{
    rhs.theSize = 0;
    rhs.head = nullptr;
    rhs.tail = nullptr;
}

List & operator= ( List && rhs )
{
    std::swap( theSize, rhs.theSize );
    std::swap( head, rhs.head );
    std::swap( tail, rhs.tail );

    return *this;
}

// Return iterator representing beginning of list.
// Mutator version is first, then accessor version.
iterator begin( )
  { return iterator( head->next ); }

const_iterator begin( ) const
  { return const_iterator( head->next ); }

// Return iterator representing endmarker of list.
// Mutator version is first, then accessor version.
iterator end( )
  { return iterator( tail ); }

const_iterator end( ) const
  { return const_iterator( tail ); }

// Return number of elements currently in the list.
int size( ) const
  { return theSize; }

// Return true if the list is empty, false otherwise.
bool empty( ) const
  { return size( ) == 0; }
```

```
    void clear( )
    {
        while( !empty( ) )
            pop_front( );
    }

    // front, back, push_front, push_back, pop_front, and pop_back
    // are the basic double-ended queue operations.
    Object & front( )
      { return *begin( ); }

    const Object & front( ) const
      { return *begin( ); }

    Object & back( )
      { return *--end( ); }

    const Object & back( ) const
      { return *--end( ); }

    void push_front( const Object & x )
      { insert( begin( ), x ); }

    void push_back( const Object & x )
      { insert( end( ), x ); }

    void push_front( Object && x )
      { insert( begin( ), std::move( x ) ); }

    void push_back( Object && x )
      { insert( end( ), std::move( x ) ); }

    void pop_front( )
      { erase( begin( ) ); }

    void pop_back( )
      { erase( --end( ) ); }

    // Insert x before itr.
    iterator insert( iterator itr, const Object & x )
    {
        Node *p = itr.current;
        ++theSize;
        return iterator( p->prev = p->prev->next = new Node{ x, p->prev, p
} );
```

```
    }

    // Insert x before itr.
    iterator insert( iterator itr, Object && x )
    {
        Node *p = itr.current;
        ++theSize;
        return iterator( p->prev = p->prev->next = new Node{ std::move( x
), p->prev, p } );
    }

    // Erase item at itr.
    iterator erase( iterator itr )
    {
        Node *p = itr.current;
        iterator retVal( p->next );
        p->prev->next = p->next;
        p->next->prev = p->prev;
        delete p;
        --theSize;

        return retVal;
    }

    iterator erase( iterator from, iterator to )
    {
        for( iterator itr = from; itr != to; )
            itr = erase( itr );

        return to;
    }

  private:
    int   theSize;
    Node *head;
    Node *tail;

    void init( )
    {
        theSize = 0;
        head = new Node;
        tail = new Node;
        head->next = tail;
        tail->prev = head;
    }
};
```

```
#endif
```

// EXAMPLE PROGRAM FOR SQUARE CLASS

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;
#include <cstdlib>   // for rand(), srand()
#include <ctime>     // for time()
#include <assert.h>
#include <math.h>    // for math functions like sqrt() etc



class Square
{
public:
      explicit Square(double s = 0.0) : side{ s }
      { }

      void setSide(double s = 1.0 )
      {
            side = s;
      }
      double getSide() const
      {
            return side;
      }
      double getArea() const
      {
            return side * side;
      }
      double getPerimeter() const
      {
            return 4 * side;
      }

      void print(ostream & out = cout) const
      {
            out << "(square " << getSide() << ")" <<endl;
      }
      bool operator< (const Square & rhs) const
      {
```

```
            return getSide() < rhs.getSide();
      }

private:
      double side;
};

// Define an output operator for Square
ostream & operator<< (ostream & out, const Square & rhs)
{
      rhs.print(out);
      return out;
}

/**
* Return the maximum item in array a.
* Assumes a.size( ) > 0.
* Comparable objects must provide operator< and operator=
*/
template <typename Comparable>
const Comparable & findMax(const vector<Comparable> & a)
{
      int maxIndex = 0;

      for (int i = 1; i < a.size(); ++i)
            if (a[maxIndex] < a[i])
                  maxIndex = i;

      return a[maxIndex];
}

int main()
{
      vector<Square> v = { Square{ 3.0 }, Square{ 2.0 }, Square{ 2.5 } };

      Square test1{ 3.0 };
      cout << "test1: " <<test1 <<endl;

      cout << "Largest square: " << findMax(v) << endl;
      char c;
      cout << "Enter any char to continue : ";
      cin >> c;
      vector<Square> u(10);
      double x;
      srand(time(NULL));// seed rand() with system time
      for (int i = 0; i < u.size(); ++i) {
```

```
            x = (double)(rand() % 100); // limit data to 0 to 99.

            u[i].setSide(x);
            //u[i].print();
              cout<<u[i];
      }

      cout << "Largest square: " << findMax(u) << endl;

      cout << "Enter any char to exit : ";
      cin >> c;
      return 0;
}



/*

Many students have asked questions about implementing Operator--() using a singly
linked list.

In one method, I was getting argument type mismatch repeatedly. I avoided it by
directly using a Node pointer instead of an iterator as below. Below is one
method for implementing it. Fill in the missing code.

*/

template <typename Object>
class List
{

 ...

public:
    class const_iterator
    {
    public:

        // Public constructor for const_iterator.
        const_iterator() : theList{ init() }, current{ nullptr }
        { }

....

        const_iterator& operator-- ()
        {
            Node* ptr = this->theList->head->next;
            while (ptr->next != this->current) { // check for other boundary
```

```
conditions here
                ptr = ptr->next;
            }
            this->current = ptr;
            return *this;
        }

        const_iterator operator-- (int)
        {
            const_iterator old = *this;
            --(*this);
            return old;
        }


    protected:
        const List<Object>* theList;
        Node* current;

        // Protected constructor for const_iterator.
        // Expects a pointer that represents the current position.
        const_iterator(const List<Object>& lst, Node* p): theList{ &lst },
current{ p }{ }

        friend class List<Object>;
    };

  ...

  class iterator : public const_iterator
    {
    public:


        iterator()
        { }



        iterator& operator-- ()
        {
 ....
        }

        iterator operator-- (int)
        {
  ...
        }
```

```
    protected:

        iterator(List<Object>& lst, Node* p) : const_iterator(lst, p) {}

        friend class List<Object>;
  }

...

}
```