```
////Class: CSE 330
//Term: Spring 2014
//Instructor: George M. Georgiou
//Names: Seth Lemanek & Alden Amoranto
//Lab 4
//Title: Sieve of Eratosthenes: List Edition
//
//    simplified List class
//
//    Described in Chapter 9 of
//    Data Structures in C++ using the STL
//    Published by Addison-Wesley, 1997
//    Written by Tim Budd, budd@cs.orst.edu
//    Oregon State University
//
//
#ifndef LIST_H_
#define LIST_H_

template <class T> class Link;
template <class T> class List;

template <class T> class ListIterator {
      typedef ListIterator<T> iterator;
public:
            // constructor
      ListIterator (List<T> * tl, Link<T> * cl)
            : theList(tl), currentLink(cl) { }

            // iterator protocol
      T & operator * ()
            { return currentLink->value; }
      void operator = (iterator rhs)
            { theList = rhs.theList; currentLink = rhs.currentLink; }
      bool operator == (const iterator rhs) const
            { return currentLink == rhs.currentLink; }
      bool operator != (const iterator rhs) const
            { return currentLink != rhs.currentLink; }
      iterator & operator ++ ()
            { currentLink = currentLink->nextLink; return * this; }
      iterator operator ++ (int);
      iterator & operator -- ()
            { currentLink = currentLink->prevLink; return * this; }
      iterator operator -- (int);

protected:
      List <T> * theList;
      Link <T> * currentLink;
      friend class List<T>;
};

template <class T>
class List {
public:
            // type definitions
      typedef T value_type;
      typedef ListIterator<T> iterator;
```

```
            // constructor and destructor
      List () : firstLink(0), lastLink(0) { }
      List (List<T> * x) : firstLink(0), lastLink(0) { }
      ~ List ();

            // operations
      bool empty () { return firstLink == 0; }
      int size();
      T back () { return lastLink->value; }
      T front () { return firstLink->value; }
      void push_front(T);
      void push_back(T);
      void pop_front ();
      void pop_back ();
      iterator begin () { return iterator (this, firstLink); }
      iterator end () { return iterator (this, 0); }
      void sort ();
      void insert (iterator, T);
      void erase (iterator itr) { erase (itr, itr); }
      void erase (iterator, iterator);
      void operator =(const List<T> & l);

protected:
      Link <T> * firstLink;
      Link <T> * lastLink;
};

template <class T> class Link {
private:
      Link (T v) : value(v), nextLink(0), prevLink(0) { }
      T value;
      Link<T> * nextLink;
      Link<T> * prevLink;
            // allow Lists to see element values
      friend class List<T>;
      friend class ListIterator<T>;
};

template <class T> int List<T>::size ()
      // count number of elements in collection
{
      int counter = 0;
      for (Link<T> * ptr = firstLink; ptr != 0; ptr = ptr->nextLink)
            counter++;
      return counter;
}

template <class T> void List<T>::push_front (T newValue)
      // add a new value to the front of a List
{
      Link<T> * newLink = new Link<T> (newValue);
      if (empty())
            firstLink = lastLink = newLink;
      else {
            firstLink->prevLink = newLink;
            newLink->nextLink = firstLink;
            firstLink = newLink;
            }
}
```

```
template <class T> void List<T>::push_back (T newValue)
      // add a new value to the end of a List
{
      Link<T> * newLink = new Link<T> (newValue);
      if (empty())
            firstLink = lastLink = newLink;
      else {
            lastLink->nextLink = newLink;
            newLink->prevLink = lastLink;
            lastLink = newLink;
      }
}

template <class T> void List<T>::pop_front()
      // remove first element from Linked List
{
      Link <T> * save = firstLink;
      firstLink = firstLink->nextLink;
      if (firstLink != 0)
            firstLink->prevLink = 0;
      else
            lastLink = 0;
      delete save;
}

template <class T> void List<T>::pop_back()
      // remove last element from Linked List
{
      Link <T> * save = lastLink;
      lastLink = save->prevLink;
      if (lastLink != 0)
            lastLink->nextLink = 0;
      delete save;
}

template <class T> List<T>::~List ()
      // remove each element from the List
{//Class: CSE 330
//Term: Spring 2014
//Instructor: George M. Georgiou
//Names: Seth Lemanek & Alden Amoranto
//Lab 3
//Title: Sieve of Eratosthenes

      Link <T> * first = firstLink;
      while (first != 0) {
            Link <T> * next  = first->nextLink;
            delete first;
            first = next;
            }
}

template <class T>
void List<T>::operator=(const List<T> & l)
{
    firstLink = 0;
    lastLink = 0;
    for (Link<T> * current = l.firstLink; current != 0; current = current-
```

```
>nextLink)
        push_back(current->value);
}



template <class T> ListIterator<T> ListIterator<T>::operator ++ (int)
      // postfix form of increment
{
            // clone, then increment, return clone
      ListIterator<T> clone (theList, currentLink);
      currentLink = currentLink->nextLink;
      return clone;
}
template <class T> ListIterator<T> ListIterator<T>::operator -- (int)
      // postfix form of decrement
{

            // clone, then increment, return clone
      ListIterator<T> clone (theList, currentLink);
      currentLink = currentLink->prevLink;
      return clone;
}

template <class T> void List<T>::insert (ListIterator<T> itr, T value)
      // insert a new element into the middle of a Linked List
{
      Link<T> * newLink = new Link<T>(value);
      Link<T> * current = itr.currentLink;

      newLink->nextLink = current;
      newLink->prevLink = current->prevLink;
      current->prevLink = newLink;
      current = newLink->prevLink;
      if (current != 0)
            current->nextLink = newLink;
      else
            firstLink = newLink;
}

template <class T>
void List<T>::erase (ListIterator<T> start, ListIterator<T> stop)
      // remove values from the range of elements
{
      Link<T> * first = start.currentLink;
      Link<T> * prev = first->prevLink;
      Link<T> * last = stop.currentLink;
      Link<T> * next = last->nextLink;

      if (prev == 0) {
            firstLink = next;
      }
      else if (next == 0) {
            lastLink = prev;
            lastLink->nextLink = 0;
      }
      else {
            prev->nextLink = next;
            next->prevLink = prev;
```

```
      }
      while (start != stop) {
            ListIterator<T> next = start;
            delete start.currentLink;
            ++next;
            start = next;
            }
}
#endif
```

```
//Class: CSE 330
//Term: Spring 2014
//Instructor: George M. Georgiou
//Names: Seth Lemanek & Alden Amoranto
//Lab 4
//Title: Sieve of Eratosthenes: List Edition
#include<iostream>
#include"list.h"

using namespace std;

void sieve(List<int> & values)
{
    unsigned int max = 101;
    int i;
    int array[max];
    // initalize all cells in the array
    for (i = 0; i < max; i++)
    {
      array[i] = i;
    }
     //search for non-zero cells
    for (i = 2; i*i < max; i++)
    {
      if (array[i] != 0)
      {
        for (int j = i+i; j < max; j += i)
           array[j] = 0; //multiples of i have been cleared
      }
    }

    for(int k = 0; k < max; k++)
    {
      values.push_back(array[k]);//copy array onto list
    }

}
int main()
{
    List<int> nums;
    sieve(nums);//call sieve funct to make only prime numbers appear
    List<int>::iterator i = nums.begin();
```

```
    for (i = nums.begin(); i != nums.end(); i++)
    {

         cout  << *i << " ";
    }
    cout << endl;
return 0;//end program with zero errors
}
```

```
Script started on Mon 05 May 2014 11:23:23 AM PDT
#]0;004470530@jb358-17:~/cse330/lab04##[?1034h[004470530@jb358-17 lab04]$ ./a.out
0 1 2 3 0 5 0 7 0 0 0 11 0 13 0 0 0 17 0 19 0 0 0 23 0 0 0 0 0 29 0 31 0 0 0 0 0 37
0 0 0 41 0 43 0 0 0 47 0 0 0 0 0 53 0 0 0 0 0 59 0 61 0 0 0 0 0 67 0 0 0 71 0 73 0
0 0 0 0 79 0 0 0 83 0 0 0 0 0 89 0 0 0 0 0 0 0 97 0 0 0
#]0;004470530@jb358-17:~/cse330/lab04#[004470530@jb358-17 lab04]$ exit

Script done on Mon 05 May 2014 11:23:50 AM PDT
```