CSCI-1200 Data Structures — Fall 2014
Lab 6 — Iterators & Linked Lists & Reversing Data

This lab explores the use of the STL `list` class and the use of list iterators.

Checkpoint 1

Write and test a function named `reverse_vector` that reverses the contents of an STL `vector` of integers. For example, if the contents of the vector are in increasing order before the call to `reverse_vector`, then they will be in decreasing order afterwards. For this checkpoint, use **indexing/subscripting** on the vector, not iterators (or pointers). *You may not use a second vector or array.*

The trick is to step through the vector one location at a time, swapping values between the first half of the vector and the second half. As examples, the value at location 0 and the value at location `size()-1` must be swapped, and the value at location 1 and the value at location `size()-2` must be swapped.

Write a main function to test the function you have written. The main function should (a) create a vector of integers, (b) output the contents, (c) pass the vector to the reverse function, and then (d) output the resulting vector. To help with this, you should write an additional function that prints the size and the contents of a vector (so you don't need to keep writing for loops over and over). Your main function should test special cases of empty vectors and vectors of one or two values. Then you should test "typical" cases. Be sure to also test somewhat bigger vectors with both an even and an odd number of elements.

To complete this checkpoint, show a TA the completed and correct reverse function and the test main function, and then show the TA the compilation and correct output.

Checkpoint 2

Write a new version of the reverse function named `reverse_list` that instead uses iterators to reverse the contents of an STL `list` of integers. Add code to the main program to test this. (You can start by copying much of the code and test cases you wrote for Checkpoint 1).

You may want to use a straightforward concept we did not discuss in lecture: a reverse iterator. A reverse iterator is designed to step through a list from the back to the front. An example will make the main properties clear:

```
list<int> a;
unsigned int i;
for ( i=1; i<10; ++i ) a.push_back( i*i );

list<int>::reverse_iterator ri;
for( ri = a.rbegin(); ri != a.rend(); ++ri )
  cout << *ri << endl;
```

This code will print out the values $81, 64, 49, \ldots, 1$, in order, on separate lines. Observe the type for the reverse iterator, the use of the functions `rbegin` and `rend` to provide iterators that delimit the bounds on the reverse iterator, and the use of the `++` operator to take one step backwards through the list. It is very important to realize that `rbegin` and `end` are NOT the same thing! One of the challenges here will be determining when to stop (when you've reached the halfway point in the list). You may use an integer counter variable to help you do this.

To complete this checkpoint, show a TA your debugged `reverse_list` function and ask any questions you have about regular vs. reverse iterators for lists and vectors.

Checkpoint 3

Now let's transition from the STL list to the low level manipulation of a singly linked list structure of simple `Node` objects.

Download and study the code in this file:

http://www.cs.rpi.edu/academics/courses/fall14/csci1200/labs/06_lists_iterators/lab6.cpp

You have two functions to write, `make_linked_list_from_STL_list` and `reverse_nodes_in_linked_list`. Study the example use of each in the main function. The first function takes in an STL list of integers and creates a structure (calling `new` once for each element in the input) and returns a pointer to the first `Node` in a chain of `Node` elements storing the same data. Implement and test this function using the provided `print_linked_list` function.

The second function you will write rearranges the Node objects that are created by the first function so that the data in the list now be printed in reverse order. You should do this by editing the `ptr` variable of each Node, rather than by editing the `values`. Note that this second function does not call `new`. Instead it rearranges the nodes of the provided list. Write a few additional test cases for each function to be sure that they work with input lists of different sizes.

Finally, note that we call `new` many times in this code, but we never call `delete`. Thus, this program has some memory leaks. We'll talk about this more in future lectures & labs.

To complete this checkpoint, show a TA your debugged implementation of these singly-linked lists functions.