NIU CSCI 340 GRADE-O-MATIC ASSIGNMENT

WRITING A CONTAINER WITH SUPPORT FOR ITERATORS

INTRODUCTION

This assignment is an opportunity for students to learn more about how iterators work in STL by implementing a container and its iterators such that various STL algorithms will *just work* with it.

Your task is to provide implementations for all of the methods of the `simple_linked_list` class, which is declared in `simple_linked_list.decl.h`. Your code to implement them will go into the file `simple_linked_list.h`.

NOTES

There is one file in which you should complete your work for this assignment – `simple_linked_list.h`. You should also submit a copy of your working `assign2-algos.h` from the previous assignment.

You are able to download the other files used in this project, along with a `Makefile`, from the AutoGrader. You can copy the files to a directory on `turing` or `hopper` and type `make` to attempt compilation. You should feel free to create whatever files you want to test things locally, including writing your own simple programs to test small parts on their own. I actually *encourage* you to write unit test programs for yourself, but they should not be a part of your submission.

You are expected to actually complete this assignment. If it is discovered that you cheated, whether that be by using someone else's work, or by finding a way to trick the AutoGrader into giving you points without implementing the requirements, it will be considered Academic Misconduct.

IMPLEMENTING AN STL-STYLE LINKED LIST

Although a doubly-linked list is already implemented in the STL in the form of `std::list`, and you should use it in the future when you need a linked list, there is a lot to be learned by writing your own STL-style container and watching iterator-based generic algorithms "just work" on it.

Toward that end, you will be implementing the member functions for several classes, which are declared in `simple_linked_list.decl` but that you should define/implement in `simple_linked_list.h`:

`linked_node` This is the class that handles your individual linked list nodes. It is already complete, and you don't need to implement anything for it, but a brief explanation of the members follows:

- `data` will store the value for the current node
- `next` will point to the next node in the list, or be `nullptr` if there is no such node.
- The constructor is provided for convenience, and it allows you to construct a node with the value and/or the next node pointer already set.

`simple_linked_list` This the class that provides the STL-style interface to your linked list. It is also declared in `simple_linked_list.decl.h`, and the important data members are:

- `head` - a pointer to the first node in the linked list, or `nullptr` if it's empty.
- `tail` - a pointer to the last node in the linked list, or `nullptr` if it's empty.

The only method that comes implemented is the default constructor. You will need to implement everything else (in `simple_linked_list.h`)

- Range constructor: `simple_linked_list(ITERATOR, ITERATOR)`
  - Initializes the list to a working state, then iterates over the range provided, adding each element to the linked list in order. This will only be called with valid iterator ranges.
- `empty()` method
  - This method should return `true` if, and only if, there are no nodes in the linked list.
- `begin()` method
  - This method should return a `simple_linked_iterator` that points to the head node.
- `end()` method
  - This method should return a `simple_linked_iterator` that would compare == an iterator that is off the end of the list.
- `front()` method

  - This should return a reference to the data member of the head node. Although this is obviously invalid when called on an empty list, the responsibility for checking falls upon the person using it, and you're allowed to crash when they try.
- `back()` method
  - This should return a reference to the data member of the tail node. Once again, although this is obviously invalid when called on an empty list, the responsibility for checking falls upon the person using it, and you're allowed to crash when they try.
- `pop_back()` method
  - Modify the linked list to remove the last node.
- `push_back(T)` method
  - Add a new node containing the provided value to the end of the linked list.
- `size()` method
  - Returns how many elements are currently in the linked list.
- `clear()` method
  - Remove all of the nodes from the linked list to reset it to an empty state.
- Destructor: `~simple_linked_list()`
  - This is called when your container goes out of scope, and is responsible for making sure to `delete` any `new` objects you created.

`simple_linked_iterator` This is the class for the iterators exposed by `simple_linked_list` above. The only data member it has is the `pos` value, which is a pointer to one of the `linked_node` linked list nodes used by your linked list.

You need to implement the methods so that they work properly as iterators when used in an algorithm. This will mean implementing the following methods (in `simple_linked_list.h`):

- Constructor: `simple_linked_iterator(linked_node<T> *)`
  - This one is actually already implemented, and allows you to construct the iterator already pointing to the given node.
- Preincrement: `operator ++ ()`
  - Called when you increment your iterator with ++ *before* the name, i.e. `++i`. This changes the current iterator so it points to the next node, and returns a reference to the current iterator, which will be at the new position.
- Postincrement: `operator ++ (int)`
  - Called when you increment your iterator with ++ *after* the variable name, i.e. `i++`. This also changes the current iterator so it points to the next node, but it returns another iterator that points to the original node position.
- Dereference: `operator * ()`
  - This should return a reference to the `data` member of the `linked_node` that the current iterator points to. Checking to make sure that it's okay to dereference is the responsibility of something else, so if someone tries to dereference something invalid, failure here is their fault, not yours. (You won't hear that often.)
- Comparison: `operator == (simple_linked_iterator &)`
  - This returns `true` if and only if both `simple_linked_iterators` (`*this` and the parameter) are pointing to the same node.

HOW TO SUBMIT

You must submit working versions of the following files on the AutoGrader.

- `assign2-algos.h` - The initial version of this has been provided as a convenience – you should submit the working one from the previous assignment.
- `simple_linked_list.h` - I've provided an unfinished version of this as a convenience – make sure you implement the assignment in this.

GRADING CONSIDERATIONS

- Did you submit the right files?
- Does it compile? Does it run?
- Does the output match? I have provided reference output for the driver program, so you can compare your program's output to what is expected. This output can be found in `container.refout`
- Are there any memory leaks? There is a potential, when working with linked lists, to mess things up and leak memory. There is a program called `valgrind` that can/will be used to check for that.
- Did you document your code? Documentation will be graded separately, but will be a portion of your final assignment grade.

- You need a docbox at the top of every one of the files you're required to change including:
  - Your name
  - Your zid
  - Your course section
  - A description of what the program does
- You should add a docbox for every function that you implement, explaining what it does and what each parameter is for.
- Add other comments inside your code blocks describing what you're doing and why.
- The use of `doxygen` style comments is encouraged, but not required.