CSCI 2270, Fall 2014
HW 1, SINGLY LINKED LIST OF INTEGERS
Due by Moodle Monday, September 22nd, 11:55 pm

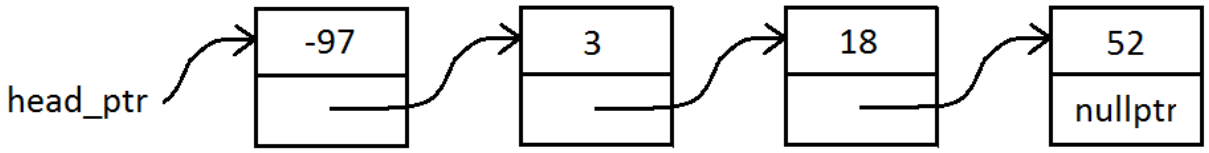
Purposes: Practice building the simplest linked list in the world, and get familiar with how to use pointers to traverse a data structure.

You will be given a header file, which does not need to change; the implementation file, which needs to be completed, and a small test file to get you started (but which is different from the complete version of the test file we'll use). Please download the most recent versions of these to get the full function set.

You are making a singly linked list composed of pointers to nodes; nodes are defined in the file singly_linked_list.h:

```
struct node {
    int data;
    node* next;
};
```

Each node contains an integer in its data field, and you will make your linked list keep these nodes in sorted order, smallest to largest, as your array did in lab 4. In the illustration below, you should remember that these arrows are pointers to the next node, meaning addresses where the next node can be found in memory:



On your VM, please make a directory for hw 1 and copy ALL the files for the linked list homework from the moodle site into that directory. As before, *you only need to change the singly_linked_list_sorted.cpp file for this assignment*. The test code contains a routine to check your linked lists for sorted order. You'll see a message if your code has an error in the sorting.

Function 1: `add_node(node*& head_ptr, const int& payload)`

Adding to a linked list in sorted order requires the following logic:

1. If the list is empty, we make a new node, set its data to payload, and set its next to `nullptr`. We then set `head_ptr` to this new node and return.

2. If the list is not empty but the new node should go before the current first node, then make a new node, set its data to payload, and set its next to `head_ptr`. Update `head_ptr` to this new node and return.
3. If the list is not empty, we want to walk the list until we find the node before where the new node should go. We make a new node, set its data to payload, set its next to the previous node's next, and set the previous node's next to the new node. Then we return.

Run the test code again and find out if your list is sorted when you add numbers to its front, middle, or end.

Function 2: `remove_node(node*& head_ptr, const int& target)`

The `remove_node(node*& head_ptr, const int& target)` code is the next part to change. This code removes one instance of the target from the list.

1. If the target is not in the list, this function does nothing except return `false`.
2. If the target is at the first node in the list, make a pointer to this soon-to-be-destroyed node, set `head_ptr` to `head_ptr->next`, and delete the soon-to-be-destroyed node. Return `true`.
3. If the target appears later in the list, then removing it involves finding the node just before the target's node in the list. Make a pointer to the soon-to-be-destroyed node. Set the previous node's next pointer to the soon-to-be-destroyed node's next pointer. Delete the soon-to-be-destroyed node. Return `true`.

Function 3: `bool find_list(const node*& head_ptr, const int& target);`

For this function, write a loop like the one in the code for `print_list`; inside your `find_list` loop, check if you have found the target and return `true` if so. When the loop is finished, if your code runs to that point, it has never found the item, so you can surely return `false`. Check that this function works correctly for integers that aren't in the list, and for integers at the front, back, and middle of the list.

Function 4: `void clear_list(node*& head_ptr);`

This function should delete every node in the linked list with no memory leaks and no segmentation faults. At the end, `head_ptr` should be equal to `nullptr`. As always, it's necessary to keep track of the remaining list nodes until you delete them too. Monday's slides have examples of how to do this.

Function 5. `void copy_list(const node*& source_ptr, node*& dest_ptr);`

This function must clear out the memory for the `dest_ptr`, using `clear()`, and then add every integer in the `source_ptr` list to the `dest_ptr` list.

Upload your singly_linked_list.cpp file to the moodle assignment link for HW 1 and make sure it is really there before you call it done.

Note: one small pointer error can potentially lose large amounts of your list. Start early and comment your code lavishly, to help you remember what it's doing. Don't be shy about looking up the LA schedules in BOLD and CSEL for help!