CIS 291 – Data Structures in C++ - Spring 2005
Week 9 Lab Exercise
Week 9 Lab Exercise due: by END of LAB on Tuesday, March 22nd

Purpose:
Thinking about destructors, copy constructors and assignment operators; more linked-list practice.

Answer the following on a piece of paper individually. Then, compare and discuss your answers with at least one other class member. Then, write your name on the "Next:" list to get your work checked over. As always, to receive credit for this lab exercise, your work must be completed by the end of the lab period.

NOTE that this time, you will lose points for incorrect answers --- I want you to be especially careful in your checking with one another, and to really discuss any differences until you have resolved them.

1. You are supposed to be reading and carefully considering the example **stack.h** and **stack.cpp** containing a linked implementation of a stack for HW #7.

One nice thing about copy constructors and destructors is that their declarations follow a very particular pattern (even if their definitions do not!).

Based on what you have read in **stack.h**, you should be able to deduce appropriate answers to the following questions:

Assume that you've written a new class **lab09thing**, with appropriate files **lab09thing.h** and **lab09thing.cpp**. Assume, also, that **lab09thing** has data fields that are dynamically allocated, so that explicitly-defined copy constructor and destructor are needed.

(a) Write the declaration for class **lab09thing**'s copy constructor as it would appear within **lab09thing.h**.

_____

(b) Write the declaration for class **lab09thing**'s destructor as it would appear within **lab09thing.h**.

_____

2. Linked list practice:

Consider our **node.h** (the one posted along with HW #7 will be fine).

Sometimes one adds additional, "helper" functions along with a node class --- they are not part of the node class itself, but they make dealing with linked structures more convenient.

The declarations for these helper functions are placed in **node.h**, after the node class declaration but before the #endif. The definitions for these helper functions are placed in **node.cpp** (or **node.template**), after the class method definitions.

Consider what should happen when you are finished with a linked list --- you want to deallocate all of the nodes, right? (Return them to the heap!) But, how are you to do that? If you call delete for

the first node, only, then that's all that you will deallocate; the rest will be left in the heap, unreachable and wasting space. There's no single delete statement that you can call that will delete the entire list.

However, one could write a node helper function that, given a pointer to a node, will "walk through" the nodes reachable from that point, carefully calling **delete** for each.

Write the **definition** for a function **list_clear**, that takes a pointer to a node as its argument, and deallocates all of the nodes reachable from that pointer.

(Note: for this lab exercise:

* you ONLY have to write this function "on paper"; you do not actually have to modify **node.h** or **node.cpp**. At least, not yet.

* no opening comment block is required for this on-paper exercise.)

BE CAREFUL --- can the address stored within the parameter pointer be changed? Should it be? So, how should this parameter be passed?

3. Now, when you have dynamically allocated data fields, in addition to an explicitly-definied copy constructor and destructor, it is also expected that one include an explicitly-defined **assignment** operator.

The good news is that the assignment operator's implementation is quite similar to the copy constructor's implementation (both are making a deep copy of dynamic data, after all).

So, the differences between the assignment operator's implementation and the copy constructor's implementation are quite small, and can be summarized as follows (adapted from Savitch and Main, "Data Structures and Other Objects Using C++", 3rd Edition, p. 179):

* The copy constructor builds a copy of an object from scratch --- the assignment operator is not constructing a new copy, but is "filling"/"resetting" an existing one appropriately.

  So, the copy constructor must allocate memory appropriately for the new copy. The assignment operator already has an allocated "destination", if you will.

  However, it is very possible, depending on the particular implementation details, that memory might still have to be newly-allocated to accomplish the assignment. For example, perhaps a dynamic array field has to be resized; and with a linked structure, there could be quite a bit of new allocation taking place. If data fields are re-allocated, then the original fields' contents need to be returned to the heap (need to have **delete** appropriately called on their behalf). (And you need to be very careful about this point when linked data structures are involved...!)

* In the assignment operator, you are expected to handle properly the case where (however goofily) one tries to assign an object to itself. (You know, **b = b**;)

  What is proper handling for this case? To simply leave the object unchanged --- to simply return at that point.

  This means every assignment operator definition should include the following code at its start (assuming that its parameter is named **source**; for a different parameter name, change the following accordingly!):

```
// Check for possible self-assignment
if (this == &source)
{
    return;
}
```

So --- now what?

Consider **stack.h** and **stack.cpp** --- currently, no assignment operator is included.

Assume that the following is added to **stack.h**, within the class declaration:

```
void operator =(const stack& source);
```

And, here is a printout of the definition of stack's current copy constructor, except with numbered lines:

```
          // copy constructor
          //
          // (adapted from list_copy, Savitch and Main,
          //    p. 241)
    1.    stack::stack(const stack& source)
    2.    {
    3.       node *source_curr, *new_curr;
    4.
```

```
 5.      top = NULL;
 6.
 7.      used = source.used;
 8
 9.
10.      // handle the case of the empty stack
11.      if (source.top == NULL)
12.      {
13.          return; // source is empty, so this copy is, too
14.      }
15.
16.      // if source is NOT empty, make a copy of it
17.
18.      // cause new copy to now have source top's data;
19.
20.      top = new node(source.top->get_data());
21.
22.      // start a pointer walking through the new copy
23.      //    starting from its top
24.      new_curr = top;
25.
26.      // is there another in the source left to be copied?
27.      source_curr = source.top->get_next();
28.
29.      // while there is another node in the source left
30.      //     to be copied...
31.      while (source_curr != NULL)
32.      {
33.          // add a new node to the new copy with the data in
34.          //    that node;
35.          new_curr->set_next(new node(source_curr->get_data()));
36.
37.          // move the new copy's pointer to the new node
38.          new_curr = new_curr->get_next();
39.
40.          // is there another in the source left to be copied?
41.          source_curr = source_curr->get_next();
42.      }
43.  }
```

Given what has been discussed above, you should be able to write the definition for operator =. Do so, on the back of this page (or on the back of any page of this lab exercise handout).

For this lab exercise:

* no opening comment block is necessary; all you have to write is the definition for operator =.

* you may assume that node.h and node.cpp now include your function from problem #2 above (hint! hint!)

* for code that would be the same as the copy constructor, you may simply write the range of line numbers in your on-paper implementation. For example,

  // lines 99-258 from copy constructor