CSCI-1200 Computer Science II — Spring 2006
Lab 14 Garbage Collection (& Class Inheritance)

In this assignment we will further explore the explicit and automatic memory management techniques discussed in lecture. Download these files from the course website and then turn off your internet connection:

`http://www.cs.rpi.edu/academics/courses/spring06/cs2/labs/14_garbage_collection/memory.h`
`http://www.cs.rpi.edu/academics/courses/spring06/cs2/labs/14_garbage_collection/memory.cpp`
`http://www.cs.rpi.edu/academics/courses/spring06/cs2/labs/14_garbage_collection/main.cpp`

In this code we have defined a base class `Memory` and two derived classes `CPP_Memory` and `StopAndCopy_Memory`. Notice how some member functions are declared and implemented as part of the base class, some member functions are only declared in the base class but must be implemented in the derived classes, and some member functions are specific to one derived class only. Logically these choices match the functionality necessary for each memory technique.

Checkpoint 1

In lecture, we stepped through the Stop and Copy garbage collection algorithm on a small example. Examine the output of the program to see a computer simulation of this same example. Verify that the program behaves as we predicted in lecture.

Continuing with the same example, 3 more nodes have been added and the garbage collector must be run again. Draw a box and pointer diagram of the memory accessible from the root pointer after these 3 nodes are added and work through the Stop and Copy algorithm for this example on paper. When you are finished, uncomment the simulation and check your work.

To complete this checkpoint: Show one of the TAs your box and pointer diagram and the scratch paper where you worked through the example.

Checkpoint 2

Write code to test the different memory error messages (NULL POINTER EXCEPTION, SEGMENTATION FAULT, CANNOT DELETE MEMORY THAT IS NOT ALLOCATED, and OUT OF MEMORY) for both derived classes.

Write a short fragment of code that runs out of memory with the `CPP_Memory` explicit memory management scheme, but *not* with the `StopAndCopy_Memory` garbage-collected memory management scheme. Then write a fragment of code that runs out of memory in *both* schemes.

To complete this checkpoint: Show one of the TAs your new test cases. Understanding the circumstances for each of these errors should help you when you need to track down memory errors in the future.

Checkpoint 3

Implement the `delete_all` helper function for `TEST_CPP_Memory`. This function should call `delete` on all nodes that are reachable from the root node. If there are no memory leaks, all the memory should be free after this call.

First implement this function using simple recursion: recurse on the child nodes and then delete the current node. Be sure to write an appropriate base case if the node is NULL. Construct a small non-cyclic data structure and verify that your code works properly. Now, try it with the large data structure in `TEST_CPP_Memory`. Why does it fail for cyclic data structures?

Rewrite the function so that it works for cyclic data structures as well. You are allowed to use `std::set`. Hint: You may want to write a helper function to do the recursion.

To complete this checkpoint: Show one of the TAs both versions of the `delete_all` function.