CSE 332 Studio Session on C++ Dynamic Memory

These studio exercises are intended to introduce basic C++ idioms for dynamic memory management, and to give you experience working with them in the Visual C++ environment.

In this studio you will again work in self-selected small groups, and as before, students who are more familiar with the material are encouraged to help those for whom it is less familiar, and asking questions during the studio sessions is highly encouraged as well.

Please record your answers you work through the following exercises. After you have finished please send your answers to the required exercises, and to any of the enrichment exercises you completed, in an e-mail to the cse332@seas.wustl.edu course e-mail account, with the subject line "Dynamic Memory Studio". The enrichment exercises are optional but are a good way to dig into the material a little deeper, especially if you breeze through the required ones. **Please make sure as you work through these exercises that each team member has a chance to participate actively – e.g., take turns coding, looking up details, debugging, etc., and please also refer to the slides and the posted code examples as you work.**

PART I: REQUIRED EXERCISES

1. Find your team members in the studio area, sit down at/around and log in to one of the Windows machines, open up Visual Studio and create a new Visual C++ Win32 Console Application project. Change the signature of the main function in the source file that Visual C++ generated to match the one that was specified for the previous studios. Write down the names of the team members who are present (please catch up anyone arriving late on the work, and also add their name) as the answer to this exercise.

2. Add a new header file and source file to your project, and in those respectively declare and define a **detector** class that you will use throughout this studio to track object creation and destruction. The class should have a private unsigned integer member variable, another private unsigned integer member variable that is static and is initialized to 0, a public default constructor, a destructor, and a public accessor method for the non-static member variable: (1) The default constructor should initialize the non-static member variable with the value of the static member variable, then increment the value of the static member variable, and print out a message (to cout) with the method name, the object's address, and the value of the initialized non-static member variable; (2) The destructor should print out the method name, object's address, and the value of the non-static member variable, to cout, and should do nothing else. As the answer to this exercise, please show the code for the class you declared and defined.

3. In your main function, dynamically allocate a detector object and use a C++11 shared_ptr to hold on to it. Add cout statements (with a closing endl to ensure the stream is flushed) to the beginning and end of your main function to mark its scope. As the answer to this exercise please explain what happens to the detector object when the shared_ptr goes out of scope.

4. Use shared_ptr variables and parameters to transfer access to the dynamically allocated detector object created in the previous exercise, from the main function into and back out of another function you write which is called by main (again use cout statements to mark the scope of that function as well). As the answer to this exercise, explain (a) which shared_ptr actually destroys the object, (b) when that happens, (c) whether the object is deleted multiple times or only once, and (d) whether or not you think any aliases to it remain after the object is destroyed and why you think that.

5. Declare several more shared_ptr variables and dynamically allocate detector objects for each of them to hold onto. Declare a vector of shared_ptr objects (to the detector object type as in the previous exercise), and push back each of the shared_ptr variables you just created into it. As the answer to this exercise, explain (a) whether or not this resulted in more, fewer, or the same number of detector objects being created and/or destroyed than in the previous exercise, and (b) why you think that happened that way.

6. Using the code from the previous exercise, obtain and print out the addresses of the objects (a) that are aliased by the original shared_ptr variables you created, and (b) the addresses of the objects that are aliased by the shared_ptr elements of the vector. As the answer to this exercise please show the output with those addresses, and explain based on them whether or not the same objects are being aliased by the two different sets of shared_ptr variables.

PART II: ENRICHMENT EXERCISES (optional, do the ones that interest you).

7. Try using a vector of pointers to detector objects that were dynamically allocated directly using the new operator. See what happens when you explicitly call delete on the pointers to the detector objects, vs. what happens when you don't. As the answer to this exercise, please explain what memory the vector cleans up for you, and what it doesn't.

8. Declare several C++ style strings in your main function, and experiment with default construction, construction with a C-style string, copy construction, and assignment. Use the strings c_str function to obtain and print out the address of the underlying C-style string (casting it to a void * when you print it out, so you can see it as a hexadecimal address). As the answer to this exercise please explain both whether or not C++ style strings are using the copy-on-write idiom, and why you think that based on what you observed.

9. Repeat the previous exercise (trying different combinations of construction and assignment of C++ style strings and looking at the addresses of the underlying C style strings) but instead of using C++ style strings as stack variables, use new and delete to do dynamic allocation and de-allocation so that you can control their lifetimes exactly. In particular, try allocating a C++ style string dynamically, use it in the copy constructor of another C++ style string that will outlive it, and then de-allocate the first C++ style string. As the answer to this exercise please explain both whether or not C++ style strings are using the reference counting idiom, and why you think that based on what you observed.