

CSE 332 Studio Session on C++ Memory Management Idioms

These studio exercises are intended to introduce basic C++ memory management idioms, 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 as you work through the following exercises. After you have finished please post your answers to the required exercises, and to any of the enrichment exercises you completed, to the course message board as a reply to my posting titled “Memory Management Idioms”. 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. Add the detector class header file and source file from the previous studio session on C++ Memory Management with Classes into your project. 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. Write a simple wrapper class that has a pointer to a detector object as its only member variable, a default constructor that dynamically allocates a detector object, a destructor that deletes the dynamically allocated detector object, and a copy constructor that uses deep copy (see previous lecture slides and studio exercises).

Implement an assignment operator using the copy constructor, as shown in the lecture slides for this studio. Add cout statements to your class methods and operators so that you can tell exactly what is happening when assignment is done. In your main function default construct, copy construct, and assign objects of the wrapper class type, and as the answer to this exercise explain both whether or not the trick for implementing the assignment operator using the copy constructor worked, and why you think that based on what you observed.

3. Use a C++ auto_ptr to hold on to a dynamically allocated detector object in your main function. 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 object when the auto_ptr goes out of scope.

4. Use an `auto_ptr` to transfer ownership of the dynamically allocated detector object created in the previous exercise, from the main function to another function 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 `auto_ptr` actually destroys the object, (b) in which function that happens, and (c) whether the object is deleted multiple times, or only once.
5. 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.
6. 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.

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

7. Try using a vector of pointers to dynamically allocated detector objects. 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. Try using a vector of `auto_ptr`s to dynamically allocated detector objects. As the answer to this exercise, please say what happens when you do that, and why that is happening (**hint:** `auto_ptr`s were not designed to work with STL containers – or vice versa).