

CSE 332 Studio Session on C++ Functions, Classes, and Templates

Before we dig into the details of C++ functions, classes, and templates over the rest of the semester, these studio exercises are intended to give an introduction to using these common (and useful) C++ features, and to give you experience using their related ideas and techniques within the Visual C++ environment.

In this studio you should again work in groups of 2 or 3, selected at will. As before, students who are more familiar with the material are encouraged to help those for whom it is less familiar. Asking questions of your professor and teaching assistant (as well as of each other) 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 “Functions, Classes, and Templates 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.

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, and write down the names of the team members who are present (if a team member arrives late please catch them up on the work, and add their name) as the answer to this exercise.
2. Open up Visual Studio, and create a new Visual C++ Win32 Console Application project for this studio. Throughout these exercises you will replace the code for previous exercises with new code – one way to do this without losing what you’ve already done (so you can refer back to it later it that’s useful) is to comment out the old code. Use the name of this project as the answer to this exercise.
3. Change the signature of the main function in the source file that Visual C++ generated to match the one that was used in the lecture slides on C++ program structure. Add a new header file and a new source file to your project. These are where you will declare and define (respectively) a struct you will develop during this studio. Use the names of the new header and source files as the answer to this exercise.
4. Declare (in your new header file) and define (in your new source file) a struct that represents a point in 3-dimensional (Cartesian) space. It should have a constructor that takes three coordinates and initializes an object using those coordinates. It should have a less-than operator (operator<) that takes a const reference to another object of the same type and returns true if and only if the current object is closer to the origin (the point <0,0,0> in 3-space) than the other object is. **Hint:** for one way to calculate this see http://en.wikipedia.org/wiki/Euclidean_distance. In your main function (**hint:** you’ll need to include the struct header file you just wrote) declare 2 objects of this type, constructed with different values, and use the less-than operator to see which is closer to the origin. As the answer to this exercise, say what the points were, and which (or say both if their distances are equivalent) was closest to the origin.

5. Extend your solution to the previous exercise by pushing back several objects of your struct type into a vector. Use the sort algorithm to sort the points (which it should do in order from closest to farthest from the origin). Iterate through the sorted vector and on a separate line for each point print out its x, y, and z coordinates. As the answer to this exercise, give the output produced by your program.

6. Repeat the previous exercise, but rewrite your struct's less-than operator so that it sorts the points first by distance from the x axis, then by distance from the y axis, and then by distance from the z axis (**hint:** see the corresponding lecture slides for an example of how to do this in 2 dimensions). The sort algorithm should now sort the points according to the new less-than operator definition you just coded. Iterate through the sorted vector and on a separate line for each point print out its x, y, and z coordinates. As the answer to this exercise, give the output produced by your program.

PART II: ENRICHMENT EXERCISES (optional, feel free to skip some and do ones that interest you).

7. Building on the previous exercise, use the next_permutation algorithm to generate, count, and print out all unique orderings of the points in the vector. As the answer to this exercise, please say how many unique orderings there were, and show the output for one of the orderings.

8. Instead of computing permutations of a vector of 3-dimensional points, have your main function take in two arguments (in addition to the program name) and wrap them as C++-style strings. Compute and print out all possible permutations of the first string, and compare each permutation to the second string – if it finds a permutation that matches the second string, it should also announce having found a match. As the answer to this exercise please answer the following questions: (1) does the program detect that “act” is a permutation of “cat” and that “lee” is a permutation of “eel”? (2) were the same number of permutations computed for “act” and “cat” vs. “lee” and “eel”, and what is a possible explanation for why or why not?