CSE 332 Second Studio Session on C++ Associative Containers

These studio exercises are intended to give additional coverage of the structure of C++ Ordered Associative Containers and the major kinds of features associated with them (and optionally to give you experience working with Unordered Associative Containers), and to give you experience using those features within the Visual C++ environment. In this studio you will again work in small groups. 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 instructors and teaching assistants (as well as of each other) during studio sessions is highly encouraged as well.

Please record your answers as you work through the following exercises. After you have finished please email your answers to the required exercises, and to any of the enrichment exercises you completed, in an e0mail to the cse332@seas.wustl.edu course email account, with the subject line "Associative Containers Studio II". 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 that as you work through these exercises that each member of your team has a chance to participate actively – one way is to take turns coding, looking up details, debugging, etc.

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 for this studio. Change the signature of the main function in the source file that Visual C++ generated to match the one that was specified in the lab assignments and in the lecture slides. 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. In your main function, declare a set container that holds C++ style strings (e.g., a container of type **`set<string>`**). Call the container's **`insert`** method repeatedly to insert different strings into the container and also try to insert a few of the strings more than once. Use **`auto`** to declare a variable to hold the value that is returned from each call and use that variable to check whether or not each call succeeded in inserting the string into the container. As the answer to this exercise, please indicate (1) what strings you tried to insert into the container, (2) which of the insert calls succeeded and which failed, and (3) how the program was able to tell which succeeded and which failed (**`hint:`** see the course text book for more about how to determine that from the variable storing the result of each call).

3. Repeat the previous exercise using a multiset container that holds C++ style strings (e.g., a container of type **`multiset<string>`**). As the answer to this exercise please describe any differences you observed compared to the previous exercise, and if there were differences please explain why you think they occurred.

4. Using the code from the previous exercise, use a string that was inserted into the container multiple times, and the container's **`equal_range`** method to obtain iterators bounding all occurrences of that same string in the container. Use those iterators with the **`copy`** algorithm and an **`ostream_iterator`** to print out the equal strings, and as the answer to this exercise please show the code you wrote to do that.

5. Starting with the code from the previous exercise, after printing out the equal range, please try the following: (1) call the container's **`erase`** method using the first iterator returned from the **`equal_range`** as function parameter (2) change the **`erase`** method to use both iterator returned from the **`equal_range`** as function parameters (3) change the **`erase`** method again to remove the copy of that string from the container. As the answer to this exercise, show the output of your program when **`erase`** is called, by invoking the container's **`equal_range`** method to obtain a fresh pair of iterators bounding all occurrences of that same string in the container, print out all the occurrences of that string.

6. Repeat the previous exercise using a multimap container that maps C++ style strings to integers (e.g., a container of type **`multimap<string, int>`**) but for each duplicate key give a different integer value that is associated with it, and when you print out the range of equal elements print both the **`string`** (i.e., the key) and the **`int`** (i.e., the mapped value). As the answer to this exercise, please (1) show the output your program produced and (2) explain based on that output in what order elements were removed from the container.

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

7. Repeat any of the previous exercises using an unordered version of the container type you used for that exercise. As the answer to this exercise, please indicate whether you saw any differences between how the program behaved with the unordered container vs. the ordered container, and if there were differences please say what they were and why you think they occurred.

8. Repeat exercise 5 or 6, but instead of obtaining fresh iterators by calling the container's **`equal_range`** method after each call to the container's **`erase`** method, store and re-use the iterators that were obtained from the first call to the container's **`equal_range`** method. Build and run your program (you may want to do this in the debugger as well as from the command terminal window) and based on what you observed, as the answer to this exercise please explain whether or not you think those iterators remained valid the entire time or if not when and how and why they became invalid and what observation you made indicates that.