CSE 332 Studio Session on C++ Sequence Containers

These studio exercises are intended to give a more complete coverage of the structure of C++ Sequence Containers and the major kinds of features associated with them, 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 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 "Sequence Containers 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 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. Back insertion sequence containers provide a **`push_back`** method that allows new elements to be added at the end of the container's range, and a **`pop_back`** method that allows the element at the end of the range to be removed from the container. Front insertion sequence containers provide a **`push_front`** method that allows new elements to be added at the beginning of the container's range, and a **`pop_front`** method that allows the element at the beginning of the range to be removed from the container.

   In your main function, declare variables of each of the following parameterized sequence container types: **`vector<int>`**, **`deque<int>`**, **`list<int>`**, and **`forward_list<int>`**. Build your solution, fixing any errors or warnings that occur (add appropriate **`#include`** and **`using`** directives to your main source file as necessary). Then try using each the above methods on each of the container variables in your main function, fixing errors and warnings where possible, and where methods simply are not supported commenting out the appropriate lines of your program. As the answer to this exercise, say whether each of the containers allows back insertion, front insertion, neither, or both.

3. Use any appropriate method whose name starts with **`push_`** that is supported by the container (per the results of the previous exercise) to push the same set of values into each of the containers, so that the values end up in the same order (from beginning to end of the container's range) in each of the containers. For each of the containers, write a **`for`** loop whose iteration variable (1) is of the container's associated **`iterator`** type (for example, **`deque<int>::iterator`**), (2) starts at the beginning of the container's range (as given by the container's **`begin()`** method), and (3) until it moves past the last element of the range (by reaching the position given by the container's **`end()`** method) prints out the value at each position and then moves to the next position. Build the program, and fix any errors or warnings that you encounter. As the answer to this exercise, (1) give the output that is produced by these loops, when you run the program, and (2) for containers that support both **`push_front`** and **`push_back`** say what effect using one push method vs. the other has on the order in which the values appear in the container.

4. Starting with the code from the previous exercise, move the entire **`for`** loop for one of the container types into a function that takes a reference to a const instance of that container type, and in the main function pass the container into a call to that function you just implemented. Try to build your program using the same (non-const) **`iterator`** type that was used in the previous exercises, in that function. Then, modify the **`iterator`** type (and anything else that needs to be changed) to allow that function to compile and correctly iterate through and print out the contents of the container. As the answer to this exercise, please describe all of the changes you had to make, in order for the function to compile and run correctly.

5. A random access container allows any position in its range to be accessed in constant time using the subscript operator (represented by square brackets **`[]`**, and also known as the indexing operator). For each of the containers, add another **`for`** loop whose iteration variable is of type **`unsigned int`**, and try to use that variable, along with the subscript operator and the container's **`size()`** method, to implement the loop so that it produces the same result as the corresponding loop from the previous exercises. Build your solution, and fix any warnings and errors you can, but then comment out any of the **`for`** loops that simply are not supported by the corresponding container's interface. Run the program and confirm that the loops that are supported produce the same output as those from the previous exercise. As the answer to this exercise, say which of the containers are random access containers, and which are not, and why you think that in each case.

6. Try copy constructing an instance of each of the container types from the previous exercise, using the previously existing instance of the same container type as the argument to the constructor. Then try constructing instances of the different container types using existing instances of each of the different other container types as the argument to the constructor. Iterate through and print out the contents of each of the containers that you were able to construct successfully, and compare their contents to the contents of the containers from which they were constructed. Based on what you observed, as the answer to this exercise please describe briefly which combinations of container types could be constructed from each other, and to what extent the contents of the existing instance were copied over into the newly constructed one.

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

7. Try the previous exercises using an **`array<int>`** and as the answer to this exercise please describe (1) to which of the other container types is an **`array`** most similar and why you think that; and (2) any differences you observed between an **`array`** and that other container type.

8. Try the previous exercises using a **`string`** and as the answer to this exercise please describe (1) to which of the other container types is a **`string`** most similar and why you think that; and (2) any differences you observed between a **`string`** and that other container type.

9. Try any of the previous exercises using const iterators vs. non-const iterators. As the answer to this exercise please describe when const iterators and/or non-const iterators could be used, versus when they could not, and for each case please explain briefly why they could or could not be used.

10. For any of the exercises that uses insertion (e.g., push_back) instead try using emplacement. For each case you tried, describe whether or not using emplacement worked, and why you think that did or did not work.