

CSE 332 Studio Session on C++ Standard Template Library Containers

These studio exercises are intended to introduce information and techniques for containers in the C++ Standard Template Library (STL), and to give you experience using them in the Visual C++ environment. Details about these containers can be found using the Visual Studio help utility (for example, search for **STL map** to find an entry describing the interface for that container) and from the links in the page at <http://www.cppreference.com/wiki/>

In this studio you will again work in 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 “STL 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 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. List the names of your team members 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: `deque<int>`, `list<int>`, and `vector<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 push method 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. A Random Access Container allows any position in its range to be accessed in (amortized) constant time using the indexing (square bracket `[]`) operator. For each of the containers add another `for` loop whose iteration variable is of type `unsigned int`, and use that variable (along with the indexing 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 exercise. Build your solution, fix any warnings and errors you can, but comment out any of the `for` loops that 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.

5. In your main function, declare *associative* container variables of type `set<int>` and `multiset<int>`, and use their `insert()` methods to add the same integer values to both of those containers that you added to the sequence containers in exercise 3 above. **Then, insert those same integer values into both containers a second time.** Write `for` loops to print out the contents of each of the containers, as in exercise 3. Build your program, fixing any errors or warnings you encounter, and compare the output from each of these containers to each other and to the output from the containers in exercise 3. Try varying the order in which you input the values to the containers – does this affect the order in which the values are stored by the containers? As the answer to this exercise, describe how `set` and `multiset` differ from each other, and how they both differ from the sequence containers, based on the output produced by your program in these cases and in the previous exercises.

6. In your main function, declare *associative* container variables of type `map<int, const char *>` and `map<const char *, int>`. Following the example shown in the STL map insert method web page at <http://www.cppreference.com/wiki/stl/map/insert/>, pass the result of calling the `make_pair()` function with the appropriate arguments (for example, 3 and “three” vs. “three” and 3) to the containers’ `insert()` methods to add the same `int/const char*` vs. `int/const char*` pairs (but with the order reversed) to both of those containers. Write for loops to print out the contents of each of the containers, using iterators and the `iter->first` vs. `iter->second` syntax shown in the example at <http://www.cppreference.com/wiki/stl/map/insert/>. Build your program, fixing any errors or warnings you encounter, and compare the output from each of these containers to each other and to the output from the containers in exercises 3 and 5. Try varying the order in which you input the values to the containers – does this affect the order in which the values are stored by the containers? As the answer to this exercise, describe how the two different `map` containers differ from each other, and how they both differ from the sequence containers and from `set` and `multiset`, based on the output produced by your program in these cases and in the previous exercises.

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

7. Repeat exercise 5, but give a comparison functor to the constructor of each of the containers. Use the STL functors described at <http://www.cppreference.com/wiki/stl/functional/> to start with, and then if you’d like you can declare and define your own functors as we described briefly in the introductory classroom discussion on the STL. Print out the contents of the containers using different functors, and as the answer to this exercise describe how the order of the elements in the container is affected in each of those cases.

8. Repeat exercise 6, but provide different key comparison functors to the constructors of the `map` containers. Again start by using the STL functors described in the functional header file web page at <http://www.cppreference.com/wiki/stl/functional/>, and then if you’d like you can declare and define your own functors. Print out the contents of the `map` containers using different functors, and as the answer to this exercise describe how the order of the elements in the container is affected in each of those cases.