

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

These studio exercises are intended to introduce information and techniques for iterators in the C++ Standard Template Library (STL), and to give you experience using them in the Visual C++ environment. Details about specific kinds of iterators can be found using the Visual Studio help utility and from the links in the page at <http://www.cppreference.com/wiki/stl/iterators/>

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 Iterators 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. 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 variables of each of the following parameterized *sequence* container types: `deque<int>`, `list<int>`, and `vector<int>`. Push back the same set of five unique integer values into each of these containers, and using iterators as in the previous studio session on STL containers write a for loop for each of the containers to print out its contents. Build and run your program fixing any errors or warnings.

Then re-factor your code so that the logic of the for loops is replaced by calls to a *single* print function template with a single type parameter (for the iterator type) that takes two iterators by value and performs a single iterative loop to print out the elements in the range from the first iterator up to but not including the second iterator. Call the print function using each container's iterators, and verify that the program's output is the same as it was when the for loops were separate. As the answer to this exercise, give the output your program produced.

3. Using the code from the previous exercise, position an iterator two positions from the start of the container's range and print out that value, in the most efficient manner possible. For random access containers this can be done in constant time using pointer arithmetic expressions on the iterators returned by the container's `begin()` and `end()` methods, but for other containers you will get an error message if you try to use pointer arithmetic (try this), and instead you must use other operators. Call the print function you developed in the previous exercise to print out the value at that position (**hint:** position an iterator just after that position as the second iterator for the call to the print function). As the answer to this exercise, explain how you were able to accomplish this in the most efficient manner for each of the different containers' specific kind of iterator.

4. Write a function template with a single type parameter (for the iterator type) that takes two iterators by value and exchanges the values at the positions pointed to by the iterators. **Hints:** (1) the function name should be something other than `swap`, which is already overloaded in the STL and can cause ambiguous call errors if you use it, and (2) you'll need to use a temporary value of the iterator type's associated `value_type`. Call the function template you created in this exercise using the iterator positions from the previous exercise for each container. Then, call the print function you wrote previously, using each container's `begin()` and `end()` iterators to show the container's contents after values at different positions were exchanged. As the answer to this exercise, give the output your program produced.

5. Ranges are defined from the first iterator position given, up to but not including the second iterator position given. Think about how you would represent an empty range using two iterators, given that assumption for how ranges are defined. Using iterators that point to valid positions in your containers, call the function templates you created in the previous exercises with different empty ranges. As the answer to this exercise, please explain what happens when you do that.

6. Write a function template that takes two iterators of the parameterized type and uses only increment (`++`) decrement (`--`) and equivalence (`==`) operators and the exchange function you wrote previously, to transpose the order of elements in a range, and that works for lists as well as for vectors and deques. **Hints:** (1) watch out for empty ranges, (2) the second iterator passed in will be just past the end of the range of elements you want to transpose, (3) the first iterator can move forward and the second iterator can move backward to work towards each other. As the answer to this exercise, give the contents of each of the containers before and after you call this function template to reorder each of them.

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

7. Write a function template (**hint:** name it something other than `binary_search`, which is overloaded in the STL which can cause ambiguous call errors) that takes two iterators and uses pointer arithmetic expressions to do binary search (assumes the range is sorted in non-decreasing order). Include the `<algorithm>` library, and call the `sort` algorithm on your vector's `begin()` and `end()` iterators to put it into non-decreasing order, and then use your function template to look for different values: (1) a value not in your container, (2) the smallest value in your container, (3) the largest value in your container, or (4) a value in between the smallest and largest values in your container. Show your function template's logic as the answer to this exercise.

8. Write a (necessarily less efficient) version of the function template from the previous exercise that works for `list` as well as for `vector` and `deque`. As the answer to this exercise explain how the logic had to change in this version of the function template, versus the one in exercise 7.