Problem Set 2

Due April 30, 11:59PM

We've covered a lot of pieces of the STL and this problem set should give you a fair amount of practice playing with the standard containers and iterators. As always, feel free to email me if you have any comments or questions. You also might want to have a reference handy for some of the later problems, since they explore material we didn't have a chance to cover in class.

Problem 0

Write a function `ReverseStack` which accepts an STL `stack<int>` by reference and modifies the `stack` so that the elements are in the reverse order as in the original.

Problem 1

The Fibonacci sequence is the sequence whose first two elements are 0, 1 and whose other elements are defined as the sum of the previous two elements. For example, the first few terms of the Fibonacci sequence are given by 0, 1, 1, 2, 3, 5, 8, 13, ...

One possible generalization of the Fibonacci sequence is to consider what happens when we change the first two terms of the Fibonacci sequence to values other than 0 and 1. For example, if we set the first two terms to -2 and 3, we get the sequence -2, 3, 1, 4, 5, 9, 14, ... Similarly, if the first two terms are 1 and -1, we get the sequence 1, -1, 0, -1, -1, -2, -3, -5, ...

Write a function `GenerateGeneralFibonacciNumbers` which accepts as input a `vector<int>` by reference and the first two terms of the generalized sequence, then populates the `vector` with the first twenty terms of the generalized sequence. *(Note: The Fibonacci sequence grows very quickly – the fortieth term is in the billions – and when working with generalized sequences the values can grow even more rapidly. If the values in the sequence grow too large, you might experience an* integer overflow*, where the values in the series are too large to fit into an* `int`*. You do not need to worry about this case.)*

Problem 2

Write a function `CycleQueue` which accepts as input an STL `queue<int>` by reference and an integer value, then rotates the elements in the queue by that many positions. For example, if the initial `queue` contained the values 1, 2, 3, 4, 5 and we cycled the elements twice, we would end up with the queue containining 3, 4, 5, 1, 2. Similarly, cycling the values 1, 2, 3, 4, 5 five times would result with the elements being in the same order. You can assume that the number of iterations is nonnegative.

Next, write `CycleVector` and `CycleDeque` functions which behave identically to `CycleQueue` except which work on a `vector<int>` and `deque<int>`, respectively. Which of the two was easier to write? Which is more efficient?

Problem 3

As we covered in class, the deque outperforms the vector when inserting and removing elements at the end of the container. However, the vector has a useful member function called reserve that can be used to increase its performance against the deque in certain circumstances.

The reserve function accepts an integer as a parameter and acts as a sort of "size hint" to the vector. Behind the scenes, reserve works by allocating additional storage space for the vector elements, reducing the number of times that the vector has to ask for more space. Once you have called reserve, as long as the size of the vector is less than the number of elements you have reserved, calls to push_back and insert on the vector will execute more quickly than normal. Once the vector hits the size you reserved, these operations revert to their original speed.*

Write a program that uses push_back to insert a large number of strings into two different vectors – one which has had reserve called on it and one which hasn't. The exact number and content of strings is up to you, but large numbers of long strings will give the most impressive results. Use the clock() function exported by <ctime> to compute how long it takes to finish inserting the strings into each of the vectors; consult a reference or the time trial code from lecture for more information on how to do this. Report the difference in insertion time between the vectors.

Problem 4

Write a function CountLetters that accepts as input an ifstream and a map<char, int> by reference, then updates the map such that each character that appears at least once in the file is mapped to the number of times that the character appears in the file. You can assume that the map is initially empty. As a useful FYI, unlike the CS106 Map, if you access a nonexistent key in the map using the bracket operators, the value associated with the key defaults to zero. That is, if you have a map<string, int> named myMap that doesn't contain the key "STL", then myMap["STL"] will yield zero (and also add "STL" as a key). Also note that to read a single character from a file, it's preferable to use the get() member function from the stream rather than the stream extraction operator >> since get() does not ignore whitespace.

Problem 5

Write a function SetSortVector that accepts as input a vector<int> by reference and returns a sorted version of the vector. While there are several ways you could implement this function (including using the STL sort algorithm, which we'll cover next week), please implement this function by putting each of the vector elements into an STL multiset, then copying them back in sorted order. Why does this require us to use a multiset rather than a regular set?

Problem 6

An important STL container that we did not have time to cover in lecture is the list. Like vector and deque, list is a sequence container, meaning that it stores elements in a specific order. It also supports the familiar sequence operations such as push_back, push_front, size, clear, empty, etc. However, unlike vector and deque, it is illegal to use the bracket syntax to access elements of a list. In fact, the only way to access list elements is to iterate over the list using iterators.

---

* For those of you familiar with big-O notation, calling push_back n times always takes O(n) time, whether or not you call reserve. However, calling reserve reduces the constant term in the big-O to a smaller value, meaning that the overall execution time is lower.

The main advantage of the `list` over `vector` and `deque` is that you can insert and remove elements from a `list` at any position very quickly (in O(1) time) because the `list` is backed by a doubly-linked list. Thus if you plan on using a `vector` or `deque` to store a sequence of elements and rarely use the bracket syntax to access individual elements, the STL `list` might be a better choice of container.

Write a program that creates an STL `list` of the first 100 integers, then prints each of them to `cout`. This is not meant to be particularly challenging, but rather to serve as a warm-up for the next problem.

Problem 7

Another interesting aspect of the `list` container is that unlike `vector` and `deque`, the `list` natively supports a wide variety of useful operations. For example, the `list`'s `reverse` member function can automatically reverse a list of elements, while the `remove` member function can efficiently remove all copies of a specific value from the `list`.

Along the lines of the `SetSortVector` function from before, write a function `ListSortVector` that sorts an STL `vector` by putting its contents into an STL `list`, using the `list`'s `sort` member function to sort the elements, then copying the values back into the `vector`.

Problem 8

How long did this problem set take you? How hard was it? Did the questions help you get a better understanding of the material? Any suggestions for future problem sets?

Deliverables

To submit the assignment, email any files you've created to htiek@cs.stanford.edu. If possible, please send all answers either in a plain-text format (C++ source files are plain text) or as a PDF.

Good luck!