Name: Soham Thaker SID: 011-748-159 Email: [sdthaker1@myseneca.ca](mailto:sdthaker1@myseneca.ca)

The highlight of this workshop was using STL functions from algorithm and numeric library. Both these libraries provide templated functions that work on any STL collections by using the built-in iterators for the respective collections. Most of these functions accept a begin and an end range of the collections for which the operation is to be performed as its first 2 parameters with exception for some which take more parameters. Another key parameter is a unary/binary(depending on the function and the overloads of it) predicate/function also known as comparator or operation(depending on the function and the overloads of it). This parameter is usually a lambda, but we can also pass a functor or the function pointer to it which must meet the definition as stated according to the C++ standard. The function(lambda/functor/function pointer) is used to perform an operation, written by the programmer on each element of the collection.

I will explain all the STL functions that I used in this workshop and the reason why I chose them over others.

for\_each() - I used this algorithm function to iterate over the collection one element at a time and applied a user defined lambda on each element in the collection. The reason why I chose it is because I had to print all the Songs in the collection and this algorithm function provides a functionality where I can apply the lambda on each element, since it passes each element to the lambda as it iterates over the collection and in that lambda, I can perform an operation like printing the state of the element passed, which is exactly what I wanted.

accumulate() – I used accumulate to accumulate a value to a variable by iterating over the collection and accumulating a specific value that is present in each collection’s element’s state. Accumulate accepts 4 arguments, first 2 as range of the collection, 3rd an initial value which will be used by the algorithm to accumulate the value as it iterates over the collection, & 4th a function where I can specify what operation needs to be performed. The lambda accepts 2 parameters, 1st an accumulated value and 2nd an element from the collection. As the algorithm iterates over the collection it accumulates the value into a local variable and returns that variable once it ends iterating over the collection which is perfect for me since I had to find the total listening time for all the songs in the collection. The return value I get is used to find the total listening time in HH:MM:SS format.

sort() – I used sort to sort the collection’s elements depending on the arguments which are passed from main to the instance function defined in SongCollection class. I used std::list as the underlying type for my collection of songs which has a limitation in terms of the operators supported for underlying iterator in std::list. And this limitation makes std::list not work with std::sort so I had to use sort function provided by std::list. The idea is the same, std::sort accepts a range and a lambda which specifies how the data is to be sorted whereas, std::list sort doesn’t need the range since its already working on a given list instance, all I have to provide std::list sort is a function that accepts 2 arguments that are of same generalized type passed to std::list at the time of instantiation. Inside the function I can write my custom logic to sort the data and that is exactly what I wanted to achieve my end goal.

transform() – Transform is one of the best algorithms if a programmer wants to update the state of the generalized type of same collection or a different collection as it iterates over the collection element by element. This is exactly what I wanted since I had to update the state of each Song’s album if the value is “[None]” to an empty string. To go about writing the logic to achieve the result, I used 4 parameter overload version for transform that accepts a range of input iterators from where the data will be queried per iteration, an output iterator where the updated data is to be stored back and lastly a function which defines the query along with return value. The input and output iterators were from the same collection since I had to update the state of a given element from the same collection to an empty string if the query returns true. If the query returns true, then I update that specific element’s state and return that element back to the algorithm which then updates the output collection which also happens to be the same as input iterator.

any\_of() – This algorithm function returns true if there is at least one element present in the collection which is exactly what I wanted since I had to find an artist passed from main to the instance function at which point, I had to query if the artist exists in the entire collection’s generalized type’s state. It accepts 3 parameters, a range of collection to iterate over and a function to apply the query to. All I have to do is write a predicate which returns a Boolean value that checks whether the artist passed into instance function is the same as the one that was received as an argument to the predicate. The algorithm itself returns a Boolean value so I can directly return the value from the instance function returned by the algorithm.

count\_if() – This algorithm function counts the number of elements that satisfy the predicate condition. I used this function to find how many artists(received as an argument in instance function) were present in the collection. The reason why I used this function is that I had to return a locally created object of type std::list from the function and before anything can be copied into that local list object it needs to be allocated some memory. So, I used this algorithm to count how many artists are present with the same name as passed into the instance function and then store the returned value into a local size\_t variable. This helped me initialize just enough memory required to be allocated to the locally created list object. If I allocate less memory, then a crash can happen when copying into this object and if I allocate more memory then unused memory can be wasted which is why I allocate just enough memory needed to hold a whole new collection of Songs.

copy\_if() – This algorithm is used to copy from a range of input iterator to the output iterator when a condition is satisfied for the predicate. This algorithm was perfect for the functionality that I needed because I had to copy to the locally created list object, those elements whose artist inside the input collection matches with the artist received as an argument into the instance function. Since the predicate returns a Boolean value, if the condition is matched then the algorithm will copy that element into the output collection. Once the algorithm ends iterating over the input collection, I return the local list object whose elements are all the songs that are from same artist.

There are numerous advantages to use STL algorithms over user defined loops. Some of them are,

\* They avoid accidental copying since they prefer move operations over copy operations to improve performance and its functionality is optimized to provide better performance.

\* They also provide much less overhead than compared to user defined functions since they primarily operate on iterators which provide faster access to the elements stored in a collection.

\* They provide better error handling than the user defined loops.

\* They are implemented by people who are good at developing algorithms and the functions have been tested and used for countless times which in turn provides much less error prone logic and a robust code than using a user defined function.

\* They are implemented in a way where they know exactly what operation can be performed on a container.

\* We can chain operations with an algorithm that returns a value.

\* They are general purpose library functions(templates) which help us pass any type of the collection as long as the collection meets the iterator operators used in the algorithm.

\* The name of these functions make code more readable for someone else who goes through the code rather than using the user defined logic since only option for the programmer is to add a comment next to the logic explaining what the function does which can create ambiguity in a lot of cases.

\* One of the greatest functionalities that we get from STL Algorithms is that we can parallelize these algorithms -- each iteration can be executed on different threads. Let's say our vector has 10,000 elements, that would take a long time to search for something using find\_if, so it splits up those iterations over several threads automatically. No need to manage extra threads manually.

3.

The sort function inside the SongCollection class can create errors if client passes an invalid field or misspells it. To go about catching errors and safely executing the logic we can create a scoped enum called Error which has 2 values, InvalidArgument & MisspelledArgument. The argument passed by the client can be checked against the values these 3 values, “title”, “album” or “length”. If the argument passed matches partially with either of 3 values, then I can return Error::MisspelledArgument or if the argument passed doesn’t match with the 3 values at all then I can return Error::InvalidArgument at which point client code has access to the enum and can query the return value of that function to determine if the previous operation of sorting the collection was successful or not.