This lecture discusses vectors, a data structure in the Standard Template Library (STL) of C++. Vectors are similar to arrays but have the advantage of dynamic resizing, meaning their size does not need to be specified at the beginning and can be expanded as needed. The lecture covers basic vector operations, such as declaring a vector, accessing its elements using index-based or range-based for loops, and finding the maximum and minimum elements using STL functions like \texttt{max_element}.

The lecture also explains the use of iterators, which allow traversal of a container without knowing its underlying data structure. For example, to find the maximum element in a vector, iterators pointing to the beginning and end of the vector are used. The \texttt{all_of} function is introduced, which checks if all elements in a vector satisfy a given condition, such as being even, using either a lambda function or an external function.

Sorting a vector is another important operation discussed, which enables efficient binary search with a time complexity of \(\mathcal{O}(\log n)\) compared to a linear search on an unsorted vector with a time complexity of \(\mathcal{O}(n)\). The lecture also covers modifying vectors by adding elements to the back using \texttt{push_back} and removing elements from the back using \texttt{pop_back}.

The lecture highlights that when objects are added to a vector, a copy of the object is made rather than the object itself. This is important to remember, especially when dealing with complex objects. An example is given where a class \texttt{B} is used to create objects, and these objects are pushed into a vector, resulting in copies being made.

Finally, the lecture discusses using vectors as a map to calculate the frequency of ASCII characters in a text. By iterating through the text and converting characters to their integer ASCII values, the frequency of each character can be tracked. The lecture concludes by noting that while vectors support additional operations like inserting or erasing elements in the middle, these operations are less efficient compared to adding or accessing elements at the back or a specific index.