STL Reading Assignment

Haoyong Ouyang

These three tutorials serve as a guide for C++ programmers to understand and effectively use the STL (Standard Template Library) for more efficient and concise coding. I focused on A Modest STL Tutorial and Power up C++ with the Standard Template Library Part One. STL is highlighted as a powerful tool for simplifying C++ programming through generic programming principles, which enable code reuse and efficiency. These tutorials focus on mastering STL containers, iterators, and algorithms, providing a foundation for solving complex problems in an easy way.

Summary of Tutorials

In these tutorials, the authors introduce the function and application of STL:

- **Containers**:
  These are data structures like vectors, lists, deques, sets, and maps that manage collections of objects. The tutorials explain the use cases for each, emphasizing the dynamic nature of vectors, the unique key-value mapping of maps, and the efficiency of sets for membership checks. They highlight methods like `push_back` for vectors and various ways to iterate through these containers.
- **Iterators**:
  Central to STL's flexibility, iterators act as generalized pointers, allowing traversal through elements in containers without exposing the underlying structure. Different iterator types, such as input, output, forward, bidirectional, and random access, enable various levels of access and traversal. Examples include using `begin()` and `end()` for iteration, and how iterators integrate with algorithms like `sort` and `copy` .
- **Algorithms**:
  STL provides a range of algorithms, such as `sort` , `find` , `min_element` , and `max_element` , designed to work with iterators. These algorithms abstract common tasks like searching, sorting, and modifying containers, making code cleaner and more efficient.
- **Functors**:
  Function objects are discussed as a way to pass operations into algorithms, allowing customized behavior during tasks like iteration or data transformation. The tutorials provide examples of creating custom function objects for specific operations and highlight the use of predefined ones like `for_each` .
- **Practical Examples**:
  Throughout the tutorials, examples illustrate transitioning from traditional C++ code to STL-based

solutions. This includes using vectors instead of arrays for dynamic sizing, employing `istream_iterator` for reading data streams, and utilizing `back_insert_iterator` for appending data without risking buffer overflows.

- **Generic Programming**:
  The tutorials emphasize STL's generic programming approach, which allows components to be composed easily while maintaining performance. This design allows algorithms to work with a wide range of container types through the application of iterators, making STL both powerful and flexible for various programming tasks.

What I Learned

Through the tutorials, I learned how to effectively apply the C++ STL in data analysis projects. I utilized STL's `vector` , `sort` , `copy` , and `remove_if` to filter and sort a list of integers.

Code:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

int main() {
    std::vector<int> numbers = {3, 1, 7, 5, 2, 8, 4, 6, 9};

    numbers.erase(std::remove_if(numbers.begin(), numbers.end(), [](int n) { return n < 5; }), nur

    // Sort the remaining elements
    std::sort(numbers.begin(), numbers.end());

    // Print sorted numbers
    std::copy(numbers.begin(), numbers.end(), std::ostream_iterator<int>(std::cout, " "));

    return 0;
}
```

To deepen my understanding, I asked ChatGPT to provide a comparable example without using STL. The example without STL was less efficient and flexible compared to using STL. This version involved manually implementing filtering and sorting using arrays, loops, and a custom bubble sort function. Through this exercise, I observed that the non-STL approach required more code to achieve the same functionality.

```cpp
#include <iostream>

// A function to sort an array using the Bubble Sort algorithm
void bubbleSort(int arr[], int size) {
    for (int i = 0; i < size - 1; ++i) {
        for (int j = 0; j < size - i - 1; ++j) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

// A function to remove elements less than a given value
int removeLessThan(int arr[], int size, int threshold) {
    int newSize = 0;
    for (int i = 0; i < size; ++i) {
        if (arr[i] >= threshold) {
            arr[newSize++] = arr[i];
        }
    }
    return newSize;
}

// A function to print the array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; ++i) {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;
}

int main() {
    int numbers[] = {3, 1, 7, 5, 2, 8, 4, 6, 9};
    int size = sizeof(numbers) / sizeof(numbers[0]);

    size = removeLessThan(numbers, size, 5);
    bubbleSort(numbers, size);
    printArray(numbers, size);
    return 0;
}
```