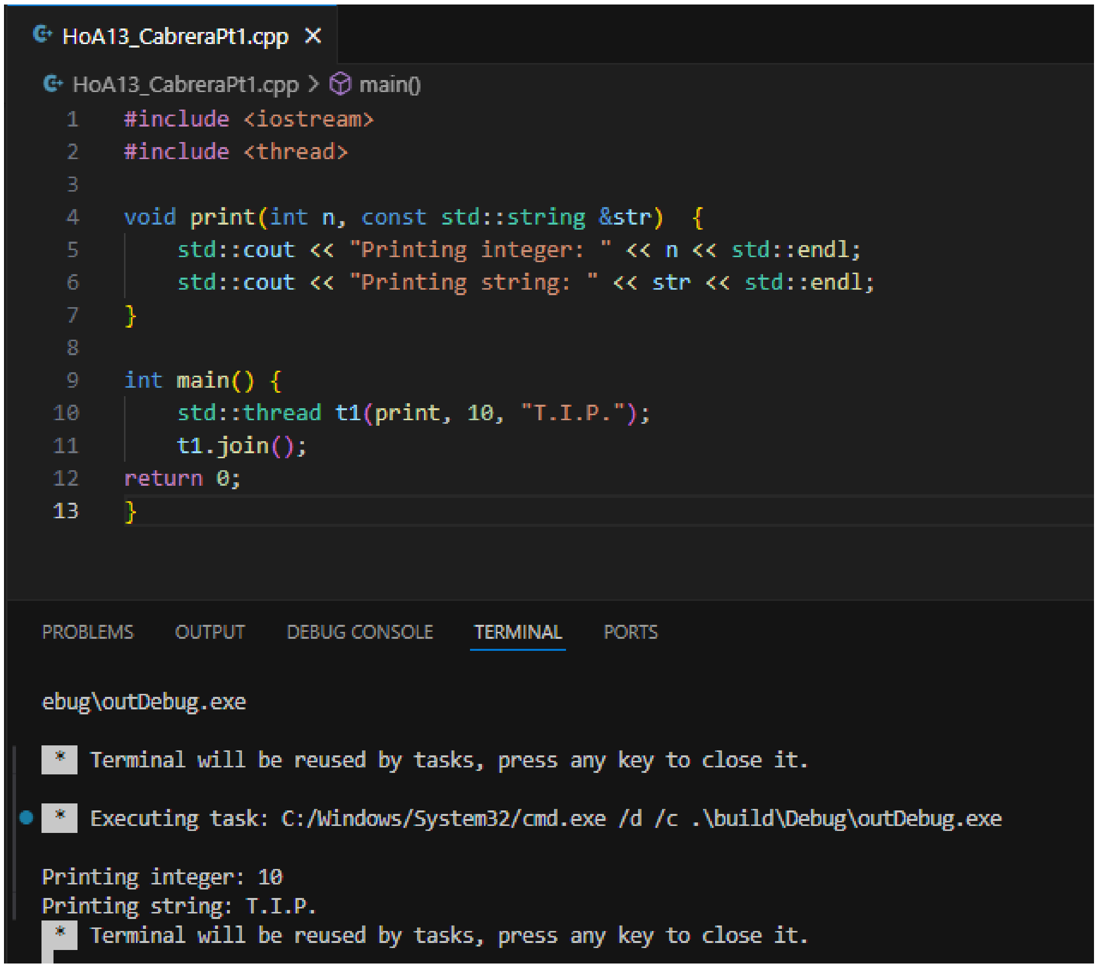
Hands-on Activity 13.1	
Parallel Algorithms and Multithreading	
Course Code: CPE010	**Program:** Computer Engineering
Course Title: Data Structures and Algorithms	**Date Performed:** 11/04/2025
Section: CPE21S4	**Date Submitted:** 11/04/2025
Name(s): Cabrera, Gabriel A.	**Instructor:** Engr. Jimlord Quejado

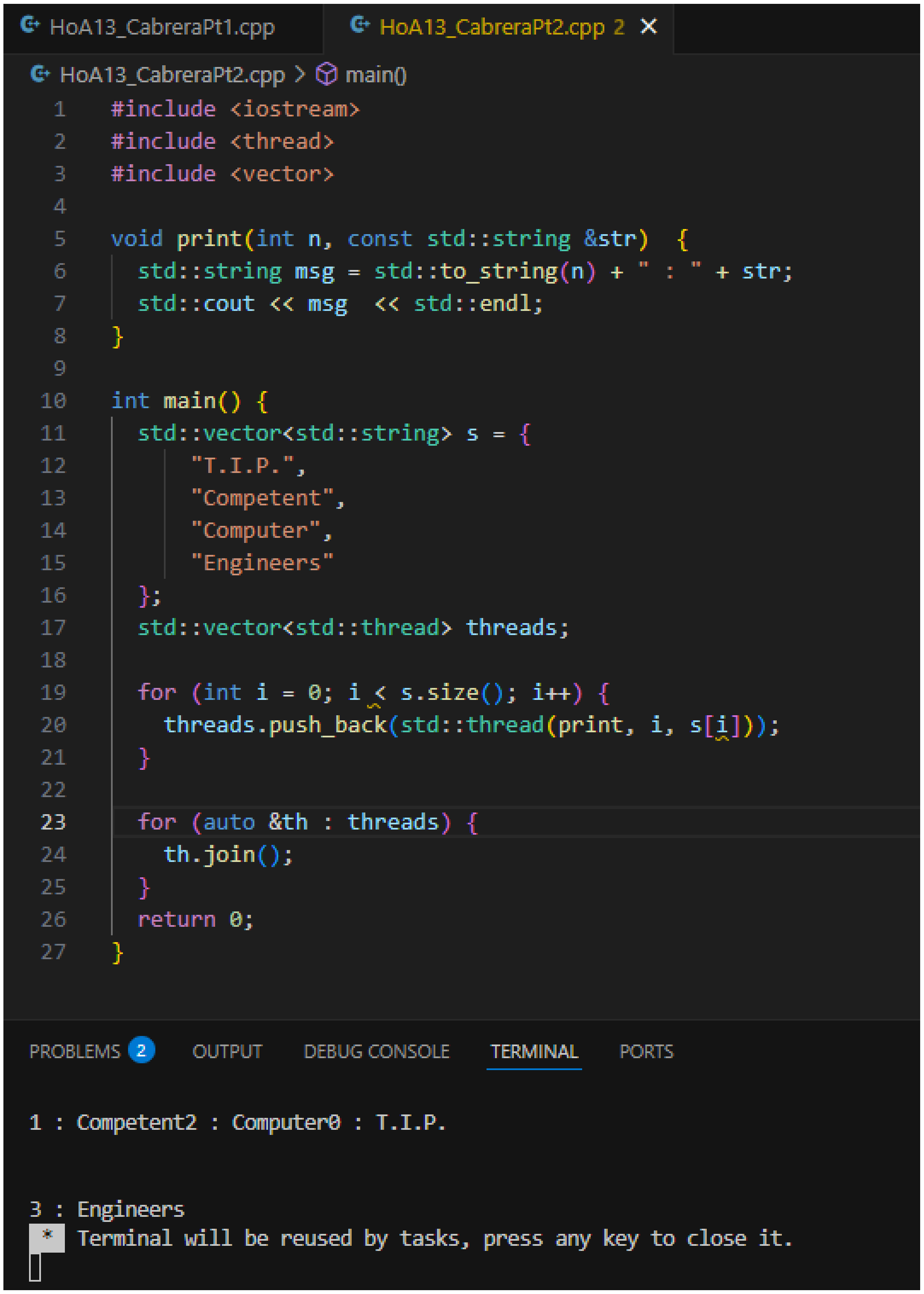
6. Output

ILO A & B:

Screenshot	 HoA13_CabreraPt1.cpp > main()
	```
	#include <iostream>
	#include <thread>

	void print(int n, const std::string &str)  {
	    std::cout << "Printing integer: " << n << std::endl;
	    std::cout << "Printing string: " << str << std::endl;
	}

	int main() {
	    std::thread t1(print, 10, "T.I.P.");
	    t1.join();
	return 0;
	}
	```
	 PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
	```
	ebug\outDebug.exe

	 * Terminal will be reused by tasks, press any key to close it.

	 * Executing task: C:/Windows/System32/cmd.exe /d /c .\build\Debug\outDebug.exe

	Printing integer: 10
	Printing string: T.I.P.
	 * Terminal will be reused by tasks, press any key to close it.
	```

Analysis	This simple program shows the use of only a single thread by creating a separate thread to execute a function that prints an integer and a string, while the main thread continues execution. The print function takes a constant integer n and a constant string reference str, then outputs "Printing integer: " followed by the value of n, and on the next line, "Printing string: " followed by the content of str. In main, a std::thread object t1 is created, passing the print function with arguments 10 and the string "T.I.P.". The t1.join() call ensures the main thread waits for the child thread to complete before the program terminates. The output shows that the code worked properly by printing the integer first, then followed by the string input.

Table 13-1. Simple One-Threaded Example.

Screenshot	
Analysis	Using the concept of threads from the previous example, this program takes it to a higher with dynamic data by launching multiple threads, each printing a unique integer and a corresponding string from a vector. A std::vector<std::string> named s is initialized with four strings: "T.I.P.", "Competent", "Computer", and "Engineers". A std::vector<std::thread> called threads is used to manage multiple thread objects. The print function takes an integer n and a constant string reference str, constructs a message by concatenating n + " : " + str, and outputs it using std::cout. In main, a loop iterates over the vector size, creating a new thread for each index i that calls print(i, s[i]) and pushes it into the threads vector. After all threads are

```cpp
#include <iostream>
#include <thread>
#include <vector>

void print(int n, const std::string &str)  {
  std::string msg = std::to_string(n) + " : " + str;
  std::cout << msg  << std::endl;
}

int main() {
  std::vector<std::string> s = {
    "T.I.P.",
    "Competent",
    "Computer",
    "Engineers"
  };
  std::vector<std::thread> threads;

  for (int i = 0; i < s.size(); i++) {
    threads.push_back(std::thread(print, i, s[i]));
  }

  for (auto &th : threads) {
    th.join();
  }
  return 0;
}
```

```
1 : Competent2 : Computer0 : T.I.P.


3 : Engineers
 *  Terminal will be reused by tasks, press any key to close it.
```

	created, a range-based for loop with auto& th iterates through the vector and calls th.join() on each thread, ensuring the main thread waits for all child threads to finish before exiting. Running the program multiple times showed different order for the printing function, this shows that the multithreading function works since multiple threads work at executing the function at the same time.

Table 13-2. Multithreaded Example.

7. Supplementary Activity

Part A: Demonstrate an understanding of parallelism, concurrency, and multithreading in C++ by answering the given questions. Use of supplementary materials to support answers must be cited as reference.

Questions:

1. Write a definition of multithreading and its advantages/disadvantages.
Multithreading is a programming technique that enables a single process to execute multiple threads at the same time, where each thread represents an independent path of execution sharing the same memory space. It allows tasks to run simultaneously within the same program, improving responsiveness and resource utilization on multi-core systems. Advantages include enhanced performance through parallel execution, better CPU utilization, improved user interface responsiveness in GUI applications, and efficient handling of I/O-bound operations. Disadvantages include increased complexity in code design, potential race conditions and deadlocks, higher memory overhead due to thread management, and difficulty in debugging and testing.

2. Rationalize the use of multithreading by providing at least 3 use-cases.
Multithreading is essential in scenarios where tasks can be divided into independent subtasks. Use-cases include: a.) In web servers, multiple client requests are handled simultaneously and each request runs in a separate thread, preventing one slow client from blocking others, b.) In video rendering software, frame processing is distributed across threads to leverage multi-core CPUs, significantly reducing render time, and c.) In real-time data processing (e.g., stock market analysis), one thread collects data while another processes and visualizes it, ensuring continuous operation without delays.

3. Differentiate between parallelism and concurrency.
Parallelism refers to the simultaneous execution of multiple tasks using multiple processing units (e.g., CPU cores), achieving true parallel computation where all tasks run at the same physical time. Concurrency, on the other hand, is the ability of a system to manage multiple tasks by switching between them rapidly, even on a single core, giving the illusion of simultaneous execution. Parallelism requires hardware support and delivers actual speedup, while concurrency focuses on task management and responsiveness, often using time-sharing. A program can be concurrent without being parallel, but true parallelism implies concurrency.

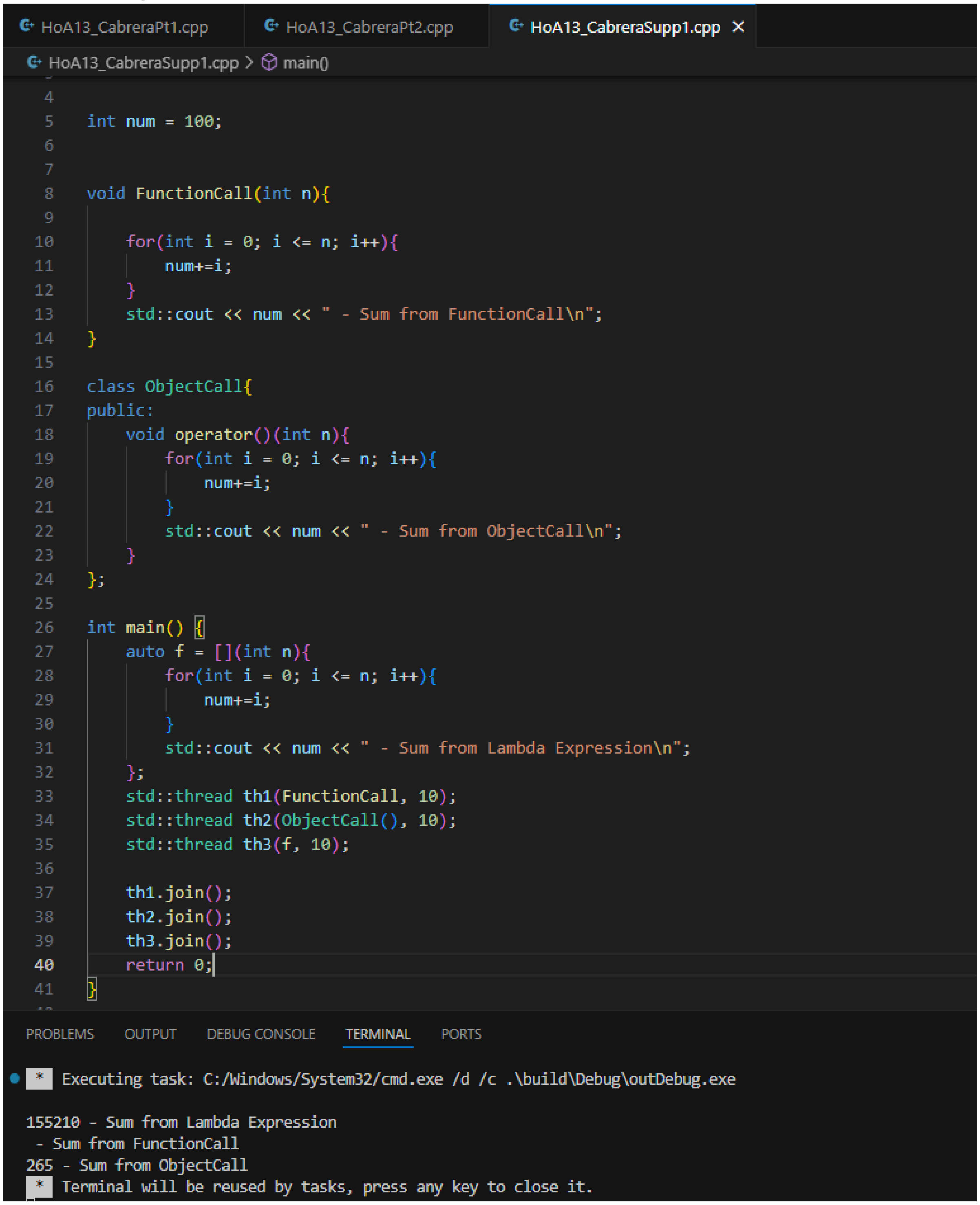
References:
C++ Concurrency in Action, Second Edition - Anthony Williams
std::thread - cppreference.com
Multithreading in C++ - GeeksforGeeks
Difference between Concurrency and Parallelism - GeeksforGeeks

Part B: Create C++ Code and show a solution that satisfies the given requirements below.

Create a global variable of type integer.
Create an add function that will take an integer parameter and add that value to the global variable.
Use multi-threading techniques to create 3 threads; individually pass the add function to the threads.
Display the value of global variable at different combinations of using the join() per thread. Such that:
o Display

o T1.join()
o Display
o T2.join()
o Display
o T3.join()
Provide an analysis based on the outputs of the multi-threading exercise.

Code and Output



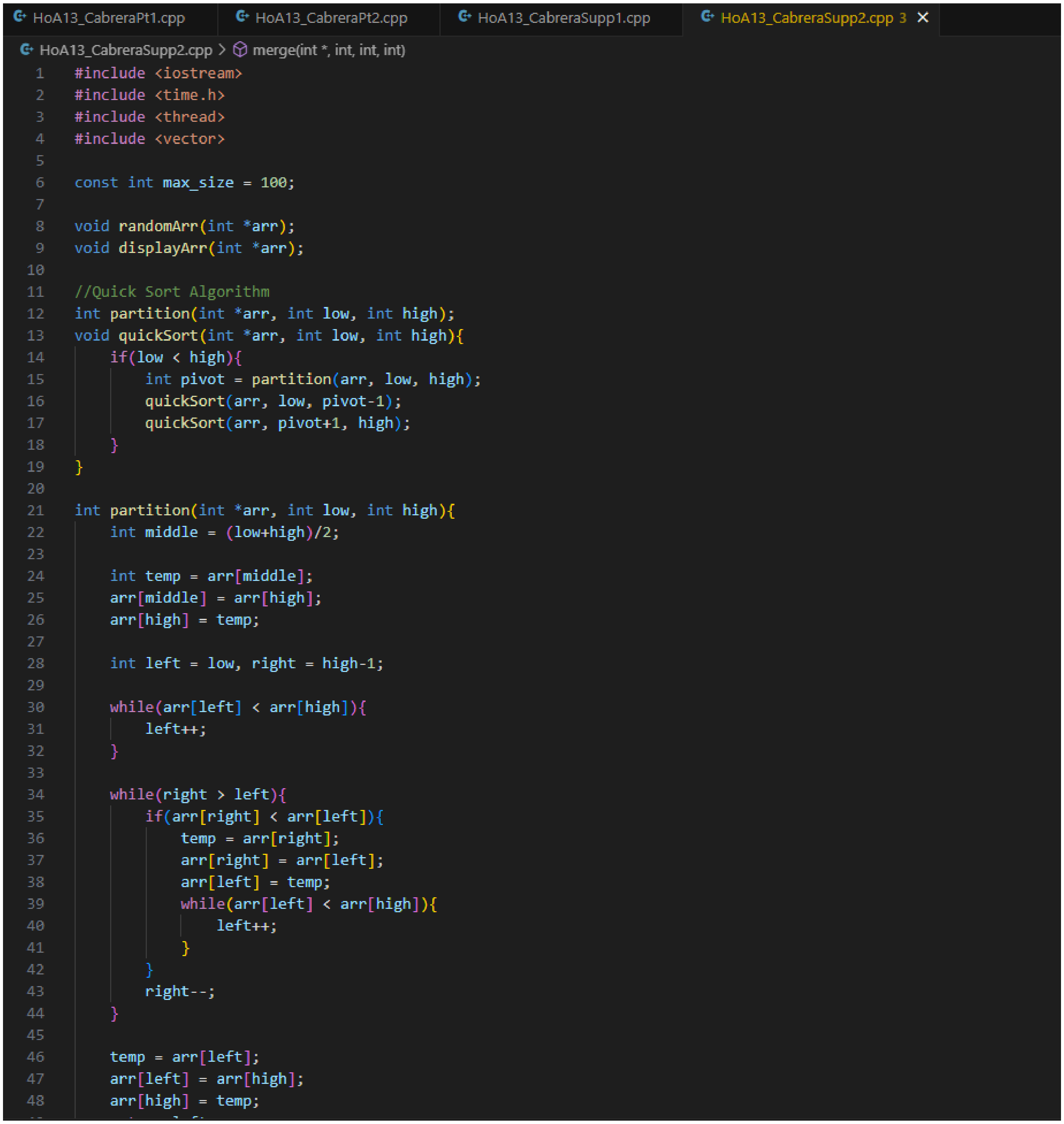
```cpp

int num = 100;


void FunctionCall(int n){

    for(int i = 0; i <= n; i++){
        num+=i;
    }
    std::cout << num << " - Sum from FunctionCall\n";
}

class ObjectCall{
public:
    void operator()(int n){
        for(int i = 0; i <= n; i++){
            num+=i;
        }
        std::cout << num << " - Sum from ObjectCall\n";
    }
};

int main() {
    auto f = [](int n){
        for(int i = 0; i <= n; i++){
            num+=i;
        }
        std::cout << num << " - Sum from Lambda Expression\n";
    };
    std::thread th1(FunctionCall, 10);
    std::thread th2(ObjectCall(), 10);
    std::thread th3(f, 10);

    th1.join();
    th2.join();
    th3.join();
    return 0;
}
```

```
 *  Executing task: C:/Windows/System32/cmd.exe /d /c .\build\Debug\outDebug.exe

155210 - Sum from Lambda Expression
 - Sum from FunctionCall
265 - Sum from ObjectCall
 *  Terminal will be reused by tasks, press any key to close it.
```

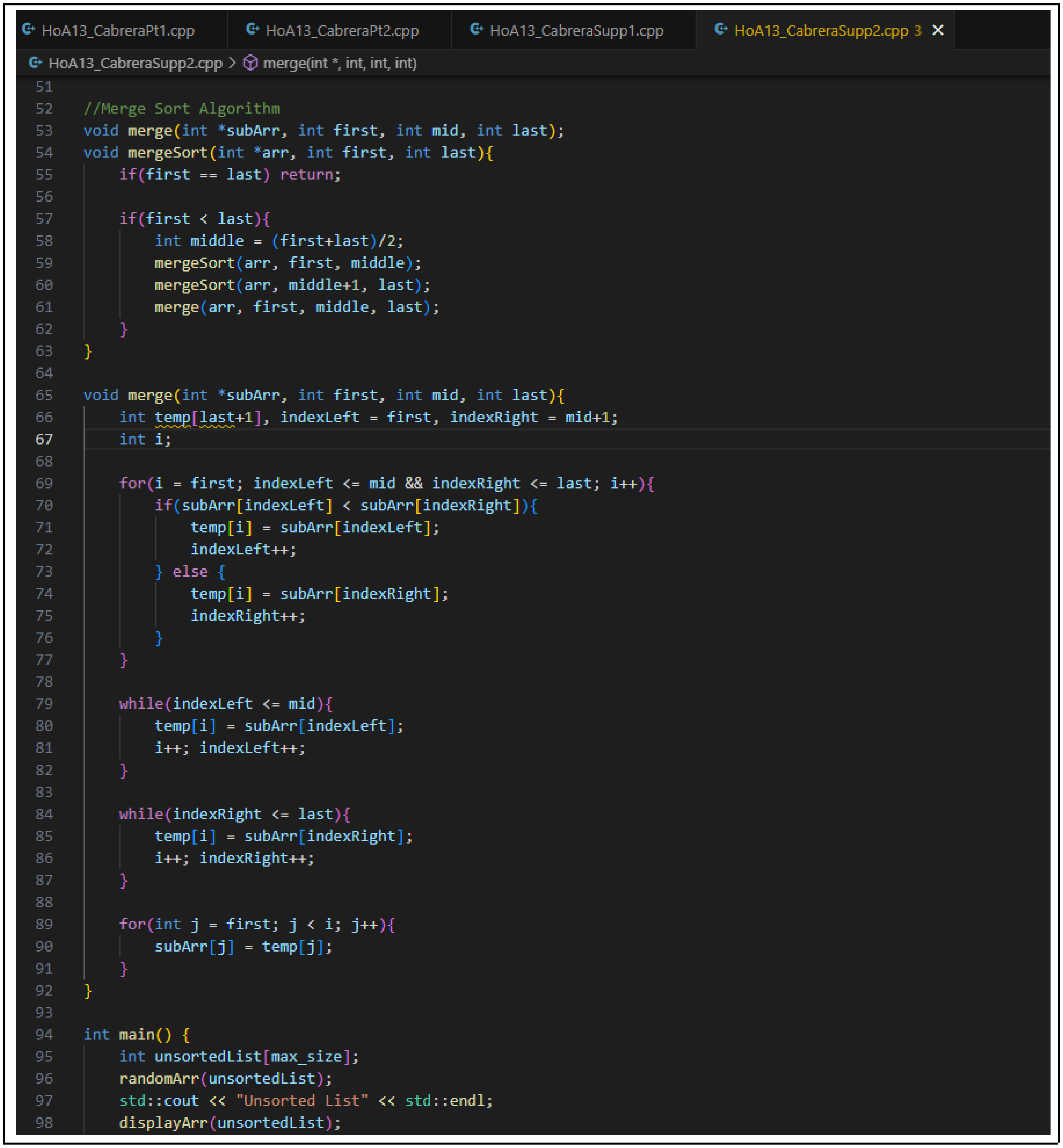
Analysis:
First, global variable num is initialized to 100, serving as the upper limit for summation. The FunctionCall function takes an integer n, iterates from 1 to n, accumulates the sum in num, and outputs the final value with the label " - Sum from

FunctionCall\n". The ObjectCall class defines a public operator() that similarly loops from 1 to n, updates num, and prints the sum with the label " - Sum from ObjectCall\n". In main, three threads are created: th1 invokes FunctionCall(10) directly, th2 uses an instance of ObjectCall() as a functor, and th3 executes a lambda expression [n](int n) that performs the same summation and prints with " - Sum from Lambda Expression\n". Each thread is started and immediately joined using join(), doing a sequential execution despite thread creation. The output looks messy since all the functions were ran and joined concurrently.

Part C: Use multi-threading with one of the algorithms previously developed in the course; provide an analysis of the result.
Code

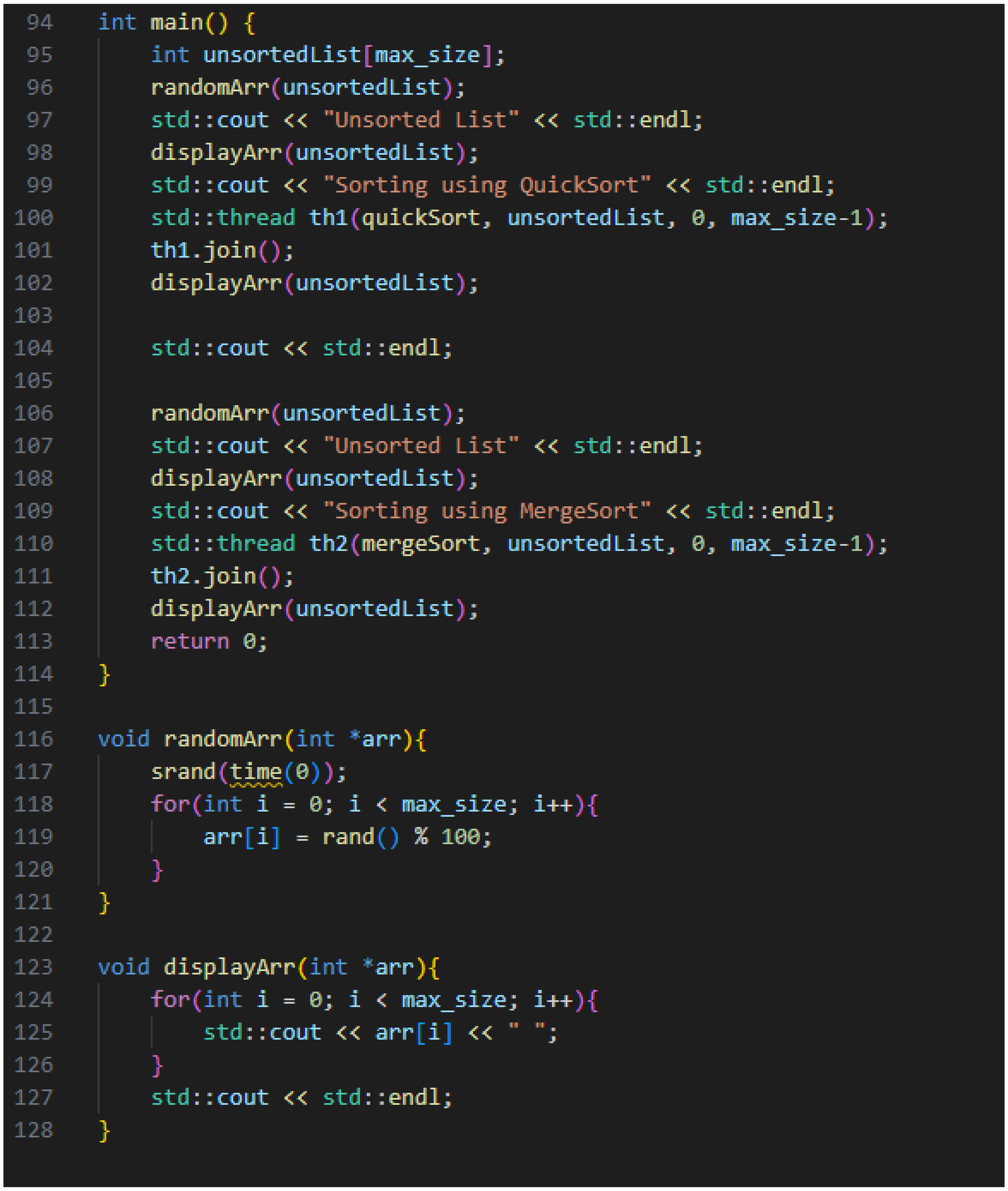


```
#include <iostream>
#include <time.h>
#include <thread>
#include <vector>

const int max_size = 100;

void randomArr(int *arr);
void displayArr(int *arr);

//Quick Sort Algorithm
int partition(int *arr, int low, int high);
void quickSort(int *arr, int low, int high){
    if(low < high){
        int pivot = partition(arr, low, high);
        quickSort(arr, low, pivot-1);
        quickSort(arr, pivot+1, high);
    }
}

int partition(int *arr, int low, int high){
    int middle = (low+high)/2;

    int temp = arr[middle];
    arr[middle] = arr[high];
    arr[high] = temp;

    int left = low, right = high-1;

    while(arr[left] < arr[high]){
        left++;
    }

    while(right > left){
        if(arr[right] < arr[left]){
            temp = arr[right];
            arr[right] = arr[left];
            arr[left] = temp;
            while(arr[left] < arr[high]){
                left++;
            }
        }
        right--;
    }

    temp = arr[left];
    arr[left] = arr[high];
    arr[high] = temp;
```


HoA13_CabreraPt1.cpp   HoA13_CabreraPt2.cpp   HoA13_CabreraSupp1.cpp   HoA13_CabreraSupp2.cpp 3

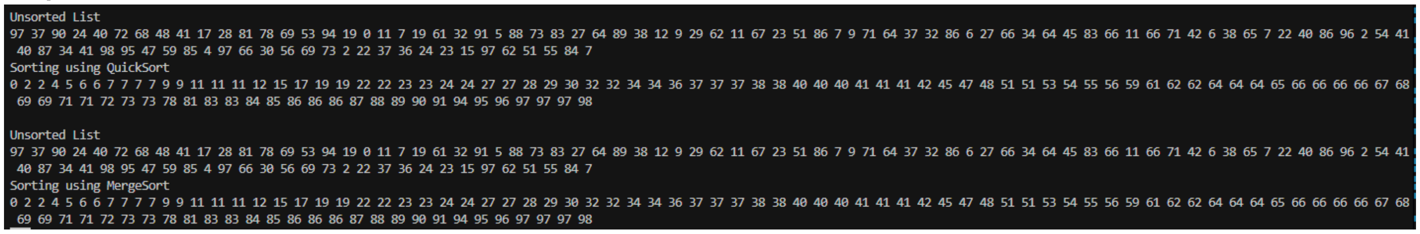
HoA13_CabreraSupp2.cpp > merge(int *, int, int, int)

```cpp

//Merge Sort Algorithm
void merge(int *subArr, int first, int mid, int last);
void mergeSort(int *arr, int first, int last){
    if(first == last) return;

    if(first < last){
        int middle = (first+last)/2;
        mergeSort(arr, first, middle);
        mergeSort(arr, middle+1, last);
        merge(arr, first, middle, last);
    }
}

void merge(int *subArr, int first, int mid, int last){
    int temp[last+1], indexLeft = first, indexRight = mid+1;
    int i;

    for(i = first; indexLeft <= mid && indexRight <= last; i++){
        if(subArr[indexLeft] < subArr[indexRight]){
            temp[i] = subArr[indexLeft];
            indexLeft++;
        } else {
            temp[i] = subArr[indexRight];
            indexRight++;
        }
    }

    while(indexLeft <= mid){
        temp[i] = subArr[indexLeft];
        i++; indexLeft++;
    }

    while(indexRight <= last){
        temp[i] = subArr[indexRight];
        i++; indexRight++;
    }

    for(int j = first; j < i; j++){
        subArr[j] = temp[j];
    }
}

int main() {
    int unsortedList[max_size];
    randomArr(unsortedList);
    std::cout << "Unsorted List" << std::endl;
    displayArr(unsortedList);
```



```cpp
int main() {
    int unsortedList[max_size];
    randomArr(unsortedList);
    std::cout << "Unsorted List" << std::endl;
    displayArr(unsortedList);
    std::cout << "Sorting using QuickSort" << std::endl;
    std::thread th1(quickSort, unsortedList, 0, max_size-1);
    th1.join();
    displayArr(unsortedList);

    std::cout << std::endl;

    randomArr(unsortedList);
    std::cout << "Unsorted List" << std::endl;
    displayArr(unsortedList);
    std::cout << "Sorting using MergeSort" << std::endl;
    std::thread th2(mergeSort, unsortedList, 0, max_size-1);
    th2.join();
    displayArr(unsortedList);
    return 0;
}

void randomArr(int *arr){
    srand(time(0));
    for(int i = 0; i < max_size; i++){
        arr[i] = rand() % 100;
    }
}

void displayArr(int *arr){
    for(int i = 0; i < max_size; i++){
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;
}
```

Output



```
Unsorted List
97 37 90 24 40 72 68 48 41 17 28 81 78 69 53 94 19 0 11 7 19 61 32 91 5 88 73 83 27 64 89 38 12 9 29 62 11 67 23 51 86 7 9 71 64 37 32 86 6 27 66 34 64 45 83 66 11 66 71 42 6 38 65 7 22 40 86 96 2 54 41
 40 87 34 41 98 95 47 59 85 4 97 66 30 56 69 73 2 22 37 36 24 23 15 97 62 51 55 84 7
Sorting using QuickSort
0 2 2 4 5 6 6 7 7 7 7 9 9 11 11 11 12 15 17 19 19 22 22 23 23 24 24 27 27 28 29 30 32 32 34 34 36 37 37 37 38 38 40 40 40 41 41 41 42 45 47 48 51 51 53 54 55 56 59 61 62 62 64 64 64 65 66 66 66 66 67 68
 69 69 71 71 72 73 73 78 81 83 83 84 85 86 86 86 87 88 89 90 91 94 95 96 97 97 97 98

Unsorted List
97 37 90 24 40 72 68 48 41 17 28 81 78 69 53 94 19 0 11 7 19 61 32 91 5 88 73 83 27 64 89 38 12 9 29 62 11 67 23 51 86 7 9 71 64 37 32 86 6 27 66 34 64 45 83 66 11 66 71 42 6 38 65 7 22 40 86 96 2 54 41
 40 87 34 41 98 95 47 59 85 4 97 66 30 56 69 73 2 22 37 36 24 23 15 97 62 51 55 84 7
Sorting using MergeSort
0 2 2 4 5 6 6 7 7 7 7 9 9 11 11 11 12 15 17 19 19 22 22 23 23 24 24 27 27 28 29 30 32 32 34 34 36 37 37 37 38 38 40 40 40 41 41 41 42 45 47 48 51 51 53 54 55 56 59 61 62 62 64 64 64 65 66 66 66 66 67 68
 69 69 71 71 72 73 73 78 81 83 83 84 85 86 86 86 87 88 89 90 91 94 95 96 97 97 97 98
```

Analysis

The program implements QuickSort and MergeSort algorithms and executes each in a separate thread to demonstrate multithreading with sorting operations on randomly generated arrays of size 100. The randomArr function seeds the

random number generator using srand(time(0)) and fills the input array with values between 0 and 99. The displayArr function prints all elements of the array in a single line.
QuickSort uses a recursive divide-and-conquer approach where the partition function selects the middle element as pivot, swaps it with the last element, and rearranges the array so smaller elements are on the left and larger ones on the right, returning the pivot's final position. The quickSort function recursively sorts subarrays on either side of the pivot.
MergeSort also follows divide-and-conquer where it recursively splits the array until single elements remain, then merges sorted subarrays using the merge function, which combines two halves into a temporary array and copies the sorted result back.In main, an array unsortedList is filled with random values and displayed.
A thread th1 is created to run quickSort on the full array (indices 0 to 99), and th1.join() ensures completion before displaying the sorted result. The array is reset with new random values, displayed again, and a second thread th2 executes mergeSort in the same manner, followed by printing the sorted output.

8. Conclusion

This activity taught me how to use modern programming techniques to implement parallel algorithms, multithreading, and concurrency. Through the procedures and supplementary activities, I have learned to use parallel algorithms, which can result in measurable performance improvements by utilizing multiple CPU cores efficiently. I also learned multithreading and concurrency concepts, these concepts enable tasks to run simultaneously, improving responsiveness and throughput in real-world applications. This practical experience has strengthened my proficiency in modern C++ concurrency features such as std::thread and std::async, even though I haven't used async at the activities.

9. Assessment Rubric