**Reflection on SFT221**

1. The most useful debugger tool was the **watch window** in visual studio community. By setting up breakpoints at critical points in the code, particularly where **isspace** and **isdigit** functions were used, and adding the **str[i]** variable to the watch window, I could inspect the values being passed to these functions. This allowed me to identify the issue with negative values resulting from char integer values being used. The very reason why the code gave me a compilation error (expression: C>= -1 && C <=255) and exited with code 3. The Watch window provided a real-time view of variable states, making it more effective than other techniques like general code inspection or static analysis, which don't offer this level of immediate insight into runtime behavior.
2. I did not use the Disassembly window or modules window of vs debugger feature. **Disassembly Window**: The Disassembly window shows the low-level assembly code generated from the source code. It is useful for examining the exact instructions being executed by the CPU, especially in performance-critical or highly optimized code. I did not use it because, the bugs were related to high-level logic and type handling in C, not issues that would require examining assembly code. The logic errors and type mismatches could be debugged effectively at the source code level. An example where I can use them is When a program behaves differently in release mode due to compiler optimizations, the Disassembly window can help pinpoint the exact instruction causing the issue.

**Modules Window**: The Modules window displays all the loaded modules (DLLs and EXEs) for the application being debugged, along with their paths, versions, and load addresses. I did not use it because the issue was confined to a single source file and its logic, with no dependencies on external modules or DLLs. Thus, there was no need to inspect module-related information. An example of where I would use it If an application is failing to load or experiencing issues due to missing or incompatible modules, the Modules window can help verify that all required modules are loaded correctly and have the expected versions.

1. Out of the three options, Using a combination of both debugger and print statements is the most efficient and effective way to find bugs. Justification: print statements are quick and easy to add to the code. They provide immediate feedback about the state of the program at specific points, helping to confirm the values of variables and the flow of execution. This is particularly useful for checking the output of loops, conditions, and the flow of the program over multiple iterations or recursive calls. A debugger allows me to pause execution and inspect the state of the program in detail. It allows me to examine the call stack, watch variables, step through the code line by line and evaluate expressions in real time. The debugger is inevitable for understanding complex bugs that involve multiple function calls or interactions between various parts of the code. A combination of both can provide the fastest debug, lets consider debugging a segmentation fault in a complex application- Using print statements to determine which function is causing the segmentation fault by printing messages at the entry and the exit of each function. Once the problematic function is identified, using the debugger to set breakpoints within the function and step into the code and inspect it line by line. Inspect pointers, check memory allocations and many more to find out exactly the line causing the fault. After making the changes we can use print statements again to double-check the flow of the program and ensure that the function now executes correctly with no errors.
2. I initially used the provided test string which consisted of multiple words, newlines, and numbers. It helped in verifying the basic functionality of line, word and number indexing. I later moved on to create additional test data covering a variety of edge cases and typical scenarios. Test data, “This test has multiple whitespaces inbetween” properly handled whitespaces and indexed the strings. It also showed me what would happen if no numbers were put in the line. The number index remains empty. I also created tests that include strings close to the MAX\_STRING\_SIZE, strings with special characters and punctuations, strings with only special characters and numbers, and strings consisting of only numbers. The software passed them all. The techniques I used consisted of edge cases – to challenge the programs limits, such as handling empty strings, strings with only whitespaces, and strings that approach the maximum allowed length. Diverse content – ensured that the test strings included a mix of words, numbers, special characters, and different types of whitespaces (space bar, tab, newline). Boundary conditions – focused on boundary conditions like the start and end of the string, handling of single characters, and transitions between different types of content (word to number, number to newline, newline to word etc.). In order to ensure the bug fixes, I used a debugger to step into the code with multiple breakpoints, checking the state of the variables, making sure of no memory leaks. I compared the program output for each case against expected results. Verified that lines, words, and numbers were correctly indexed and retrieved. I ran the program multiple times with same and different inputs to ensure consistent results all across the board, which indicated stability. In conclusion, a combination of using provided test data, creating additional comprehensive test cases, and employing debugging tools ensured that the bugs were identified, fixed, and verified. This methodical approach gives a high level of confidence in the robustness of the program, though ongoing testing remains essential for long-term reliability.