Bugs

1. The newlines are not being registered:

   As the program runs the conditions that check the characters are correctly recongizing that it has encounter a new line character and the correct position of it but it fails to save more than the first encountered line. This is because the size and index of the new line index array remains the same across the iteration.

2. The separation between words and numbers is wrong:

   The only string thats saving in the word is first encountered string from that point on all strings regardless of the content are stored inside the numbers index array. The function that checks if its numeric is working correctly, meaning that bug remains when calling the function because of the incorrect contents of the calculated variables.

Solution

1. Finding the way to solve the problem was simple since it only involve increasing the number of lines when ever it encountered one. For this the assignation sentence was changed to *result.numLines++* guaranteeing that all newline will be saved.

2. With the bugs location delimited to function calling instead of the inside of the *isNumber()* function. With the help of the debugger, it was seen the value of the length of the evaluated string was wring being a negative number if rest of cases this caused the function to have a predetermined output for all. To fix this the calculation of the len was changed to same value as the next white space sp with function calling looking like *isNumber(str + i, sp)*.

```
struct StringIndex indexString(const char* str)
{
	struct StringIndex result = { {0}, {0}, {0}, 0, 0, 0 };
	int i, sp;
	strcpy(result.str, str);
	if (str[0] != '\0')
	{
		result.lineStarts[0] = 0;
		result.numLines = 1;
	}
	for (i = 0; str[i] != '\0'; i++)
	{
		while (str[i] != '\0' && isspace(str[i]))
		{
			if (str[i] == '\n')
			{
				result.lineStarts[result.numLines++] = i + 1;
			}
			i++;
		}
		sp = nextWhite(str + i);
```

```
            if (isNumber(str + i, sp))
            {
                result.numberStarts[result.numNumbers++] = i;
            }
            else
            {
                result.wordStarts[result.numWords++] = i;
            }
            i += sp - 1;
        }
        return result;
```

Reflextion

➔ The most useful debugging technique was hardcoding, specifically incorporating the debugger inside Visual Studio. This tool allowed me to monitor the changing variables throughout the program's execution without the need to create print statements or temporary variables. The reason why this technique was more useful than others is due to how quickly it can show results regarding the internal functioning of the program. While other techniques require more planning or preparation, this one allows for an easy and in-depth examination of the code's functionality. However, even with these advantages, it's not a technique that can be used on its own; it should be accompanied by others like error limitation or code slashing. This is because, without a specific range to search, we can obtain an absurd amount of unrelated information, which can make the debugging process more challenging.

➔

- Log reporter

  This feature was not used because there was no need to monitor how the computer was registering the actions; we only needed to inspect the internal values of the variables. Since the bugs were related to logic issues, it was unnecessary to delve into detailed operation reports. These logs are better utilized when we have an application with multiple processes executing simultaneously. In such cases, examining the various processes with their runtimes and outputs provides more significant benefits.

- Memory Viewer

  Since this program doesn't involve any kind of memory manipulation or dynamic memory allocation, it doesn't provide any utility for monitoring changes in memory. However, in cases where we are implementing advanced memory management techniques to optimize heap memory usage, this feature becomes valuable for tracking memory loss. A tool like this is particularly useful when attempting to identify memory leaks in applications since they may not manifest in the contents of variables or through code errors.

- ➔ The debugger, by itself, is the faster way to look for bugs since it allows us to examine all variables and their changes across the program just by specifying breakpoints that can even be added at runtime. If we were to use print statements, we would have to add multiple lines of code throughout the program in an attempt to identify inconsistencies in the data. This represents an inefficient use of time, as a breakpoint where the print could occur is faster. If we already know where the bug is located, we could use a print statement for specific execution or testing, which may be faster than the debugger. However, in general scenarios, the debugger is better.

- ➔ In the first place, to ensure that the origin is correctly located and corrected, a testing bug was created to confirm that the root of the problem was identified. This helps in affirming when the variables are changing and how we are affecting them. Additionally, the small functions were tested individually to verify that their single function was correct by conducting some general testing. Finally, a similar string to the one provided in the main was created to test more scenarios where the code might fail. The main aspects that were tested included whether the numbers were correctly assigned regardless of their position and whether the indexes matched regardless of the length of the sub-strings or the proximity to a new line character.