1. **Problem Statement**

Create a class is a version of the string class called myString. The class should contain methods to modify, set, and display a string of characters. The outputs from the tests should be written to a text file formatted into columns.

1. **Requirements**
   1. **Assumptions**
      1. The string will be no longer than 25 characters
   2. **Specifications**
      1. The myString class should contain 14 functions. It should have a status variable that is referenced by all functions for errors.
         1. Size() – Returns how many characters are in the string
         2. addStart(mystring) – will ad the string in the input parameter to the front of the current string
         3. addEnd(myString) – will add the string in the input parameter to the end of the current string.
         4. partString(startPos, length) – returns string from startPos for length given (handler startPos < 0, startPos = size returns null string, handler startPos > size)
         5. replPartString(myString, startPos) – replaces characters starting at startPos with input string
         6. replWholeString(myString) – replaces what is currently in string with input parameter string
         7. compareString(myString) – compares the current value with the input parameter and returns a boolean
         8. initString() resets/initializes a string to null
         9. setString(string) – assign string to myString data value
         10. printStringScreen() – print the string value to the screen
         11. numericString() – is the string an integer or real(signs, decimal point, etc.)
         12. alphabeticString() – is the string all alphabetic characters
      2. The application will test the myString class by reading though multiple test files and print the results to the screen as well as to a file that corresponds to the test file being run.
         1. The format for the test file is as follows:

\*string one\*

\*string two\*

\*index one\*

\*index two\*

1. **Decomposition Diagram**

Main

Print the output and any errors to the screen for the user to view.

Write each test to a file that is formatted with into columns

Run each string through each function of myString

Input file with strings to test the myString class with

Input

Output

Process

1. **Test Strategy**

* Valid Data
* File Handling

1. **Test Plan Version 1**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test Strategy | Test Number | Description | Input | Expected Output | Actual Output | Pass/Fail |
| Valid Data | 1.0 | Test file containing 1 string with 10 characters and 1 with 15 characters. Index1 = 5 Index2 = 10 |  |  |  |  |
| Valid Data | 1.1 | Test file containing 1 string with 15 characters and 1 with 12 characters. Index1 = 14 Index2 = 7 |  |  |  |  |
| Valid Data | 1.2 | Test file containing 1 string with 10 alphabetical characters and 1 with 15 characters. Index1 = 25  Index2 = 25 |  |  |  |  |
| Valid Data | 1.3 | Test file containing 1 string with 10 numerical characters and 1 with 15 characters. Index1 = 0 Index2 = 10 |  |  |  |  |
| Valid Data | 1.4 | Test file containing 1 string with 27 numerical characters and 1 with 15 characters. Index1 = 1  Index2 = 10 |  |  |  |  |
| File Handling | 2.0 | No test files exist |  |  |  |  |
| File Handling | 2.1 | Test file is empty |  |  |  |  |

1. **Initial Algorithm**
   * 1. Create a class called myString that acts as a version of the builtin string class.
        1. In the private field:
           1. An integer variable that is called status

0 = all good

1 = out of bounds error

2 = invalid input parameter type

* + - * 1. A 25-character char array used for storing the string (empty characters will be null.
      1. Function called size – returns how many characters are in the string.
         1. Create an int variable used to store the size while counting
         2. For each non-Null value in the 25-character char array, increase size variable by 1
         3. Set status reference to 0 and return the size variable
      2. Function called addStart(myString) - Add the string in the input parameter to the start of the current string
         1. Get the size of the inputted myString value
         2. For each character in the current string

Add the character to the end of the inputted string.

If the new string size is 25 characters, discard the rest of the current string and set it equal to the newly built string

* + - * 1. Set the current character array value equal to that of the new array
        2. Set status reference to 0
      1. Function called addEnd(myString) – add the string in the input parameter to the end of the current string
         1. If the current string’s length is less than 25 characters

For each character in the inputted string

Add the character to the end of the current string array

If the current string size equals 25 characters, then discard the rest of the inputted string characters

* + - * 1. Set status reference to 0
      1. Function called partString(startPos, length) - returns string from startPos for length given (handler startPos < 0, startPos = size returns null string, handler startPos > size)
         1. Create a new myString object
         2. Check the length parameter exceeds the length from the startPos to the end of the current string

If it exceeds

Set status reference to 1 and return null

* + - * 1. If startPos is greater than current string length, set status reference equal to 1 and return null
        2. If startPos is less than 0, set status to 0 and return a null string object
        3. Create a character array that contains the number of characters specified in the length parameter
        4. For each element in the current array starting at startPos and ending at length

Set the value in the character array of size length equal to the current character in the current string array

* + - * 1. Set the string of the new myString object equal to the new character array.
        2. Return the new myString object and set status reference to 0
      1. Function called replPartString(myString, startPos) – replaces characters starting at startPos with myString
         1. If startPos is greater than the length of the current string set status reference to 1 and return null
         2. if myString parameter length is greater than the current string length-startPos, set status reference to 1 and return null
         3. starting at startPos, for each character in current string and starting at 0, for each character in myString

set the indexed character in the current string equal to that of the indexed character in the myString input parameter

* + - * 1. set status reference to 0 and return the new current string
      1. Function called replWholeString(myString) – replaces the current string value with the new myString value
         1. replace each character in the current string with the corresponding chacter in the myString parameter value
         2. set status reference equal to 0 and return the new current string
      2. Function called compareString(myString) – compares the current string value with the myString input parameter
         1. For each character in the current string and myString parameter

If the two referenced characters do not equal one another

Set status reference to 0 and return false

* + - * 1. Return true
      1. Function called initString() – resets/initializes string to null
         1. For each value in the current myString character array

Set the indexed value equal to null

* + - * 1. Set the status reference equal to zero
      1. Function called setString(string) – assign string to myString data value
         1. Create a character array that is length of the inputted string parameter
         2. Copy each character of the inputted string parameter to the character array.
         3. For each character in the 25-character myString array,

Set the indexed myString character value equal to the corresponding value in the new character array. (disregards all characters past the 25th value)

* + - * 1. Set status reference to 0
      1. Function called getString() – returns string of data from the myString data value
         1. Set status reference to 0 and return the 25-character array
      2. Function called printStringScreen() – prints the my string data value to the monitor
         1. Print an empty line to the screen
         2. For each character in the 25-character array

If the indexed character is not null

Print the character to the screen

* + - * 1. Print a new line to the screen
        2. Set status reference to 0
      1. Function called numericString() – determine if the string is an integer, float, or double (using lexical\_cast to attempt to convert to the different data types)
         1. If current character can be converted to an integer, set status to 0 and return true
         2. If current character can be converted to a double, set status to 0 and return true
         3. If current character can be converted to a float, set status to 0 and return true
         4. Else set status reference to 0 and return false
      2. Function called alphabeticalString() – determine if the string contains only alphabetical characters
         1. For each character in the current string

If the indexed character is <= z & >= a or <= Z & >= A

Set status to 0 and return true

* + - * 1. Set status to 0 and return false
    1. Main function – called when the application initially starts and will be used to test the myString class
       1. User will input the directory that contains all of the test files
          1. Test files are named myStringTest\_x\_x.txt (x\_x corresponds to the test number)
       2. User will input a directory to place the output log files
       3. For each file in the test directory
          1. Create an output text file in the directory specified by the user
          2. Initialize two myString objects: myStringOne and myStringTwo
          3. Set each set the strings values of those classes equal to the two strings in the file

Using the myString.setString(string) function

Get the string values using getString()

Print to screen and write to output file formatted in columns using logResults() function (“Action: setString ” “Original value:” \*myString Original Value\* “Parameters: ” “ New Value: ” \*new myString value\* “Returned: “ “Status: ” \*myString status\*) – new line

* + - * 1. Get the size of myStringOne using getSize()

Get the string values using getString()

Print to screen and write to output file formatted in columns using logResults() function (“Action: getSize ” “Original value:” \*myString Original Value\* “Parameters: ” “ New Value: ” \*new myString value\* “Returned: “ \*size of string\* “ Status: ” \*myString status\*) – new line

* + - * 1. Add myStringTwo to the start of myStringOne using addStart(myStringTwo)

Get the string values using getString()

Print to screen and write to output file formatted in columns using logResults() function (“Action: addStart ” “Original value:” \*myStringOne\* “Parameters: ” \*myStringTwo\* “ New Value: ” \*new myStringOne value\* “Returned: “ “ Status: ” \*myString status\*) – new line

* + - * 1. Reset myStringOne and myStringTwo back to the original values
        2. Add myStringOne to the end of myStringTwo using addEnd(myStringOne)

Get the string values using getString()

Print to screen and write to output file formatted in columns using logResults() function (“Action: addEnd ” “Original value:” \*myStringTwo Original Value\* “Parameters: ” \*myStringOne\* “ New Value: ” \*new myStringTwo value\* “Returned: ““ Status: ” \*myString status\*) – new line

* + - * 1. Reset myStringOne and myStringTwo back to the original values
        2. Get part of myStringOne starting at index1 and ending at index2 using partString(index1, index2)

Get the string values using getString()

Print to screen and write to output file formatted in columns using logResults() function (“Action: getPart ” “Original value:” \*myStringOne Original Value\* “Parameters: ” \*index1, index2\* “ New Value: ” \*new myStringOne value\* “Returned: “ “ Status: ” \*myString status\*) – new line

* + - * 1. Reset myStringOne and myStringTwo back to the original values
        2. Replace part of myStringOne with myStringTwo starting at index1

Get the string values using getString()

Print to screen and write to output file formatted in columns using logResults() function (“Action: replPartString ” “Original value:” \*myStringOne Original Value\* “Parameters: ” \*myStringTwo, index1\* “ New Value: ” \*new myStringOne value\* “Returned: “ “ Status: ” \*myString status\*) – new line

* + - * 1. Reset myStringOne and myStringTwo back to the original values
        2. Replace the entire myStringOne with myStringTwo using replWholeString(myStringTwo)

Get the string values using getString()

Print to screen and write to output file formatted in columns using logResults() function (“Action: replWholeString ” “Original value:” \*myStringOne Original Value\* “Parameters: ” \*myStringTwo\* “ New Value: ” \*new myStringOne value\* “Returned: “ “ Status: ” \*myString status\*) – new line

* + - * 1. Reset myStringOne and myStringTwo back to the original values
        2. Compare myStringOne with myStringTwo using compare(myStringTwo)

Get the string values using getString()

Print to screen and write to output file formatted in columns using logResults() function (“Action: compare ” “Original value:” \*myStringOne Original Value\* “Parameters: ” \*myStringTwo\* “ New Value: ” \*new myStringOne value\* “Returned:” \*result from compare\* “ Status: ” \*myString status\*) – new line

* + - * 1. Reset myStringOne back to null using initString()

Get the new myStringOne value using getString()

Print to screen and write to output file formatted in columns using logResults() function (“Action: initString ” “Original value:” \*myStringOne Original Value\* “Parameters: ” “ New Value: ” \*new myStringOne value\* “Returned: “ “ Status: ” \*myString status\*) – new line

* + - * 1. Reset myStringOne and myStringTwo back to the original values
        2. Determine if myStringOne is a numeric string using numericString()

Print to screen and write to output file formatted in columns using logResults() function (“Action: numericString ” “Original value:” \*myStringOne Original Value\* “Parameters: ” “ New Value: ” \*new myStringOne value\* “Returned: “ \*result from numericString\* “ Status: ” \*myString status\*) – new line

* + - * 1. Determine if myStringOne is an alphabetic string using alphabeticString()

Print to screen and write to output file formatted in columns using logResults() function (“Action: alphabeticString ” “Original value:” \*myStringOne Original Value\* “Parameters: ” “ New Value: ” \*new myStringOne value\* “Returned: “ \*result from alphabeticString\* “ Status: ” \*myString status\*) – new line

* + - 1. Function called logResults(File location, Data to log to file) – logs the requested data to the requested file
         1. Try to open the requested file

Write the data in the input parameter to a new line.

* + - * 1. Close the file stream

1. **Test Plan Version 2**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test Strategy | Test Number | Description | Input | Expected Output | Actual Output | Pass/Fail |
| Valid Data | 1.0 | Test file containing 1 string with 10 characters and 1 with 15 characters. Index1 = 5 Index2 = 10 | “abcde12345”  “zxyed1234567890”  5  10 | All functions will succeed |  |  |
| Valid Data | 1.1 | Test file containing 1 string with 15 characters and 1 with 12 characters. Index1 = 14 Index2 = 7 | “12345abcdefghik”  “abcde1234567”  14  7 | All functions will succeed |  |  |
| Valid Data | 1.2 | Test file containing 1 string with 10 alphabetical characters and 1 with 10 characters. Index1 = 25  Index2 = 25 | “abcdefghij”  “0123456789”  25  25 |  |  |  |
| Valid Data | 1.3 | Test file containing 1 string with 10 numerical characters and 1 with 15 characters. Index1 = 0 Index2 = 10 | “1234567890”  “abcdefghijklmno”  0  10 |  |  |  |
| Valid Data | 1.4 | Test file containing 1 string with 27 numerical characters and 1 with 15 characters. Index1 = 1  Index2 = 10 | “abcdefghijklmnopqrstuvwxyz1”  “1234567890abcde”  1  10 |  |  |  |
| File Handling | 2.0 | No test files exist | Test file directory contains no files | Program will end |  |  |
| File Handling | 2.1 | Test file is empty | Test file containing no data | Program will move to the next test file and ignore the empty one |  |  |
| File Handling | 2.2 | Test file data is in an invalid format | Test file containing data in an invalid format | Output an error message and skip to the next test file |  |  |

Part 1 ends here!!!!!!

1. **Code**

Copy and paste your code here. MAKE SURE TO COMMENT YOUR CODE!

A baseline for commenting is before any function add this:

//Description: What does the function do

//Pre-condition: What do input do you need for the function to work

//Post-condition: What is the end result of the function or what do you get out of the function

Also the beginning of your program should have these comments:

//Program Name:

//Programmer Name:

//Description:

//Date Created:

1. **Updated Algorithm**

Copy and paste Initial Algorithm and make any updates to reflect the changes you made in your code. HIGHLIGHT THE CHANGES YOU MAKE! Strike out deleted statements. Any statements that just have a wording change – make change and highlight (i.e. no need to strike out individual word changes). This is the FINAL documentation of your program and needs to match what code you created.

1. **Test Plan Version 3**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test Strategy | Test Number | Description | Input | Expected Output | Actual Output | Pass/Fail |
| Copy and Paste from Version 2 | Un-highlight any test cases are from version 2 that were highlighted |  |  |  | What was the actual output from your code | If your actual output matches with expected output, write PASS otherwise write FAIL |
|  | Add any final test cases you could think of and HIGHLIGHT THEM |  |  |  |  | Any test cases that fail you must change your code to make the cases pass |

1. **Screenshots**

Screenshots of your testing goes here. YOU MUST HAVE A SCREENSHOT FOR EVERY TEST CASE. A screenshot may picture multiple test cases. For each screen shot caption it with a list of the test cases are depicted in it.

1. **Error Log**

Any issues you had while testing your code are recorded in the error log as you perform testing of the “completed” code – that is, when you run through all of the test cases in the test plan.

|  |  |  |
| --- | --- | --- |
| Error Type | Cause of Error | Solution to Error |
| Log 2 types of errors:  Logic  Runtime | What specifically caused the error to occur | What did you do/change to fix the error |
|  |  |  |

Do not list any syntax errors or errors detected in unit testing as you build your program.

1. **Status**

What is the final status of your program? Does it fully work? Are there any test cases that fail and if so which ones? What needs to be done to correct the defects?