

HW: Mars Rover (MyString)

Overview

Objectives

- Practice writing classes with dynamic memory
- Practice operator overloading
- Apply the Rule of Three

Submission

Submit the following files to the autograder.

- MyString.h
- MyString.cpp

Allowed Includes

- `<iostream>`
- `<fstream>`
- `<tuple>`
- `<limits>`
- `<stdexcept>`
- `"MyString.h"`
- `"Rover.h"`

Introduction

On February 18, 2021, we celebrated the landing of the latest Mars rover, *Perseverance*. The [Perseverance rover](#) is a remarkable feat of science and engineering. Engineers must optimize every aspect of their design to accommodate strict requirements for space vehicles. In fact, due to reliability specifications, [Perseverance is powered by a 20 year old processor!](#)

For this assignment, assume you are on a team of engineers working on a sophisticated chemical composition analysis system for a new Mars rover. This rover will have even more advanced automation and learning capabilities than previous ones, and the chemical analysis tool is a key part of its ability to study the martian surface. The rover uses the [Simplified molecular-input line-entry system](#) (SMILES) to store and operate on chemical data. SMILES are string representations of 3 dimensional molecular structures, enabling computational software to model complex structures in a format that is easy to store and use. Your team discovers that the program isn't running correctly on the rover, and you've identified the problem is tied to memory. Specifically, the rover doesn't have quite enough memory to load the full program. Rather than remove core functionality, you've discovered that just removing some library code frees up enough resources to make the program run.

You must implement a custom `MyString` object that will mimic the standard C++ string object for certain functions. To facilitate this, we have provided a sample "test" rover which is nothing more than some simple functions to call your `MyString` object. We will read a set of basic commands from a file to simulate the

chemical composition tool. Note that you must not use the library string or vector classes since that would defeat the purpose of MyString.

Getting Started

- Download the [starter code](#)
- **Read** the provided main() in RoverTest.cpp and the simplified rover operations in Rover.h and Rover.cpp.
 - **You should not change these files.** They utilize functions that should be in the MyString class.
- Note that there are two blank files: MyString.h and MyString.cpp.
 - You will need to use these files to contain your MyString class (which you are to implement from scratch).
 - **You should write your own main function to test different parts of your MyString class.**
- Go through the [Requirements](#) below to review the functions you will implement.
 - Before you can compile, **you must write at least** the function declarations and function definitions that return some value of the expected type for non-void functions.
 - Once you have the basic setup of your class, even though it may be incomplete, you can start using local testing and Gradescope to test your code.
- Repeat the process of writing tests, implementing, testing, and submitting to incrementally develop a functionally correct MyString class.

Requirements

For this assignment, you will be writing a custom MyString class from scratch. It is based on the C++ string object, so we will be referencing the [CPlusPlus.com's string reference](#) extensively!

Note that on some reference pages, there may be multiple versions of C++ listed. You can pick the most recent standard available since we are compiling against C++17. For example, the image below shows options for C++98 and C++11, with version 11 highlighted.



General Suggestions

- A good first step is to **read the provided test rover files**. Inside each file, there is a comment of the form “// MyString: ...” which tells which operation will need to be implemented to make that part of the code work.
- When you start writing your class, remember to implement **header guards** in order to avoid compiler errors tied to repeat includes.
- The following would be a good command line to start compiling. You need to have declarations and at least a definition for each function, even if it doesn't do anything, before it will compile.

```
g++ -std=c++17 -g -Wall -Wextra -pedantic-errors -Weffc++ -Wno-unused-parameter  
-fsanitize=undefined,address RoverTest.cpp Rover.cpp MyString.cpp
```

MyString private members

The most important part of your custom MyString object will be its data. Recall that strings are essentially dynamic character arrays that also track their size and capacity. You will want to write your MyString class with private members to hold this information.

Because this part is not directly tested, you will need to implement this to a certain extent yourself. If you find yourself struggling with this part, start slowly by reviewing the key concepts you know:

How do you define a class named MyString?

How can you define private members to hold the size and capacity inside the new class?

What about the character array? Remember you'll need it to be a pointer since you want to dynamically allocate and deallocate the data on the heap based on the string's size.

Also, remember that our string object is essentially a c-string on the inside, just with additional members to track size and capacity and perform some operations. You may want to review the c-string lab or your lecture notes on the subject. **The key thing to recall is that the character array ends with a null terminator: '\0'**

MyString public members

You can find comments in the files of which functions and parameters you must implement. Below, we list the functions belonging to the original C++ object that we will require in testing your new MyString object, and these are necessary for the provided runner to work.

Note that we are using the name given by Cplusplus.com at the top next to the function signatures. Furthermore, keep in mind that references to “string” datatype will have to use your “MyString” datatype instead.

string reference

- (constructors)
 - default (1)
 - copy (2)
 - from c-string (4)
- (destructor)
- resize:
 - Top one only: `void resize (size_t n);`
 - This one isn't tested explicitly but is necessary to support many other operations
- capacity
 - Tracks the current size of the internal array
- size
 - Size is the current number of elements in the array
- length
- data
 - This function returns the internal character array's reference because it is actually a c-string on the inside, so remember the null terminator
- empty
- front
 - Second one only: `const char& front() const;`
- at
 - Second one only: `const char& at (size_t pos) const;`
 - Remember to do bounds checking and throw the exception if out of bounds
- clear
- operator<<
- operator=
 - string (1)
- operator+=
 - string (1)
- find
 - string(1)
 - If a start index isn't passed, you search from the beginning of the string, but if one is passed, you only search at and after that index in the string
 - Consider using a default parameter for the position rather than overloading the `find()` function name
 - The specification says to return `npos` if not found, but you may just return `-1` if not found; they are effectively the same

You can implement the following functions for **extra credit**:

- [operator==](#)
 - (1) (top one only)
- [operator+](#)
 - string (1) (top one only)

These helpers were **already provided**:

- [operator>>](#)
 - Our provided implementation isn't exactly the same as `operator>>` for the standard string class, but it does the main operation of consuming (but not saving) whitespace.
- [stoi](#)
 - We only include a base 10 conversion, but this one is interesting because you should recognize it as essentially the reverse of number slicing.

Testing

You should test locally! Not only are you provided an implemented main, but you can write your own to perform smaller tests on specific functions, like constructors.

`RoverTest.cpp` is a program that reads a text file with sample commands and SMILES strings. The commands are:

SCAN	- save the string to the internal saved SMILES
PRINT	- print the saved SMILES string
READ	- read a specific index of a SMILES string
JOIN	- append the SMILES string from the file to the one saved in memory
CLEAR	- clear the string
TEST	- append the SMILES string but don't save, just returns the result
FIND	- search for the SMILES compound within the one saved in memory

Sample command file input

```
SCAN  C1=C(C(C=C(C10)C1)O)C1
JOIN  [NH4+]
PRINT
CLEAR
JOIN  N+(C1=CC)
PRINT
TEST  (COP(=O)(O)OP(=O)
FIND  CC
```

Corresponding output

```
C1=C(C(C=C(C1O)C1)O)C1[NH4+]  
N+(C1=CC)  
N+(C1=CC)(COP(=O)(O)OP(=O)  
CC was found
```

All the logic for this is provided! You need only to implement the underlying MyString object to make the program work.