CS256 — Exercise 6

February 10, 2016

Due: Wednesday, February 17, 2016 before midnight (60 points)

1 Preparing the Project

1. Go to https://codebank.xyz and you should see a project in the CS256 group named `CS256-EX6`. Fork this project for your own user.
2. Next, clone the repository so you have a local copy:

   ```
   $ git clone https://codebank.xyz/username/CS256-EX6.git
   ```

3. The repository should have three files: `String.cpp`, `String.h`, and `main.cpp`.

2 The `String` class

In this exercise, we will replicate the behavior of a `String` class. That is, a class that is used to store sequences of `char`s. Our class will take some things from C++'s `std::string` and some things from Java's `java.lang.String`.

Like the previous exercise, you have been provided with three files: the class header, class implementation, and a test driver. The class header and implementation will start mostly empty. The test driver has a class that is marked a friend of `String` so it can access private members as needed.

Note: for this exercise, we will have to manage the memory for a dynamically allocated array of `char`s that will store the actual string contents. You need to be very careful with this memory management. Keep in mind that the tests may pass even if you do not properly handle the memory management so getting all tests to pass will not guarantee full credit if you make mistakes in the memory management.

1. Compile and run the provided files. You should see the first test has passed already as long as all of the files exist as expected.
2. Next, add two instance variables to `String`:
   (a) A `char*` named `data` that will store the actual string content as a dynamically allocated array.

   > You will get no credit if you use something like `std::vector` for `data`.

   (b) An `int` named `size` that will store the length of the string. Our `data` array may have extra space allocated; the `size` variable will tell us how much is actually in use.
3. Add the following constructors:
   (a) A default constructor that initializes the `String` to an empty string. Do this by *dynamically allocating* `data` with size 10 and then setting `size` to 0.

(b) A constructor that takes a single `char`. It should allocate space for 10 `char`s in `data`, put the passed `char` in to the first location, then set `size` to 1.

(c) A constructor that takes a `C-string` (in other words, a `const char*`), allocates at least enough space for the contents of the `C-string`, copies the contents to the new `String`'s `data` array, and sets `size` to the length of the C-string.

(d) A copy constructor. That is, a constructor with the following prototype:

```
String(const String& other);
```

This constructor should copy the contents of `other` to the `String` being constructed.

4. You must also implement a *destructor* that properly deallocates the `data` array.
5. Uncomment the next test function to test your constructors.
6. Overload the assignment operator for properly copying `String`s. Note, you must do proper memory management here! The object referenced by `this` should have its `data` array deallocated and reallocated based on `right`'s content[1]. Don't forget to properly handle self assignment!

   Your assignment operator should have the following prototype and should return `*this`.

   ```
   String& operator=(const String& right);
   ```
7. Uncomment the next test function to test your assignment operator.
8. Next, we will add several functions to provide some utility to our `String` class. Make sure to mark functions `const` if they shouldn't modify the object.

   (a) A `length()` function that should return the value of `size`.

   (b) A `charAt(int)` function that should return the `char` at the given index in our `data` array.

   (c) An `indexOf(char)` function that should return the *first* index where the given `char` appears in our `String` or -1 if the `char` does not appear in the `String`.

   (d) Overload the `[]` operator to allow access to the characters in the `String` for both reading and modification.
9. Uncomment the next test function and run your program again.
10. Next, we will provide two overloaded `+` operators for concatenation. Note: neither one modifies the operands; instead they return the result of the concatenation.

    (a) The first one will perform `String` concatenation and should have the following prototype:

    ```
    String operator+(const String& right) const;
    ```

    (b) The next one will allow us to append a `char` to the end of a `String` and should have the following prototype:

    ```
    String operator+(char c) const;
    ```
11. Uncomment the next test function to test your concatenation.
12. Overload the six relational operators to work for `String`s. Two `String`s should be considered equal when they have the same length and content. One `String` is less than another when it would come first alphabetically (using ASCII values instead of just letters). The prototypes for the `==` and `<` operator are the following:

[1] Note: if you want, you can also track the space allocated for each `String` in another instance variable and only reallocate if the current `String` does not have enough space to copy over the content of `right`. This would be more efficient, but makes the code a bit more complicated. This concept can be applied in several functions in this exercise.

```
bool operator==(const String& right) const;

bool operator<(const String& right) const;
```

From these, you should be able to determine the appropriate prototypes for the `!=`, `>`, `<=`, and `>=` operators.

13. Uncomment the next test function to test your relational operators.
14. Create a `substring` function that will work somewhat like Java's `substring` that takes two `int` parameters: the index of the start of the substring and the index of the position after the last character in the substring. It should have the following prototype:

    ```
    String substring(int start, int end) const;
    ```

    If the `start` index is invalid (negative or past the end of the `String`) or the `end` is less than or equal to `start` return an empty `String`. Otherwise, if the `end` is past the end of the `String`, return a substring up to the end of the `String`.
15. Uncomment the next test function to test your substring.
16. Finally, we will overload the `>>` and `<<` operators to allow `istream` and `ostream` to work with our `String` objects. These functions should have the following prototypes:

    ```
    std::ostream& operator<<(std::ostream& os, const String& s);
    ```

    and

    ```
    std::istream& operator>>(std::istream& is, String& s);
    ```

    For input, you should use the `>>` operator on the `istream` to read in to a `std::string` (that's C++'s string class) then copy its contents to your `String` object.

    These two operators should be declared *outside* of your class declaration in the header file. You should make the `operator>>` function a `friend` of `String` because it has to modify private data. They are not scoped on the class because the `String` is not the object calling them.

3 Submission

Once you have completed the program, you can use `git add`, `git commit`, and `git push` to push the changes to https://codebank.xyz. You can make as many commits and push as many times as desired until the deadline.