Once again, you will complete this lab individually. As usual, you may discuss the lab with classmates, subject to the empty hands policy.

Assignment: Write your own `String` class to mimic some functionality from the `String` class in Java. As much as possible, functionality of your methods should match the functionality provided by the `String` class in Java, so visit the Java API for a refresher on what those methods should do..

You may use the C++ `std::string` type as your instance variable, and use any methods provided by `<string>` to help in your implementation of the required functionality below[1].

```
String();
String(const String& other);
String(const char* cString);  // accepts a C-string literal, e.g., "Lilly"
~String();

std::string toString();   // returns a C++ string, not a String
int         length();

String      substring(int start, int end); // returns a String, not a std::string;
                                            // follow Java convention on parameters

bool    operator==(const String& other);   // so one can use if (str1 == str2) ...
String& operator=(const String& other);    // so one can assign str2 = str1;
String& operator+=(const String& other);   // so one can use str2 += str1;
```

Notes of interest:

- Overloaded operators, e.g., `operator==`, are covered in your text in Section 9.6.

- Unlike Lab 1, this code should be in two files: a header file called `String.h` that provides the *interface* for the `String` class, and the file `String.cpp` that contains the code that *implements* the methods (with one or two possible exceptions). `String.h` should be "# included" in your `String.cpp` file.

- A reasonable API reference for the C++ libraries can be found at `http://www.cplusplus.com/` (also linked on the course web page).

- Appropriate use of the C++ `std::string` methods will result in brief implementations for many of the methods. (In English, use the std::string methods "under the hood" of your implementation.)

- For handling invalid indices to `substring`, you should throw an out-of-range exception defined in `<exception>`. For example, you should include code similar to that below:

```
std::string msg = "Invalid arguments to substring: [" +
    std::to_string(start) + ", " + std::to_string(end) + "]";
throw std::out_of_range(msg.c_str());  // note conversion from string to const char*
```

    This will require a try-catch block in your tester — see section 5.6 on Exceptions in your text. See also `std::exception` at the C++ API, if you are so inclined.

---

[1]The interested student may also want to investigate *move constructors* and *move assignment operators* available in C++. For more information, see the C++ API or `http://bit.ly/move_constructor`. Addressing this now is not a requirement, as we will cover move constructors and assignments later in the semester

- Make sure to handle the case of a `nullptr` being passed as the C-string literal to your corresponding constructor — throw an invalid-argument exception. Note that this exception is not "user defined". Rather, like all the exceptions we will use, it is part of the standard template library.

- I will provide a tester file on the course web page. It will do some testing, but you should (must) expand it to do a more thorough job of testing. Part of the grading for this assignment will depend on the number and quality of the tests that you add.

- In your submission, include a `README.txt` file describing your tests (don't forget boundary/edge cases), their results, and how they demonstrate the correctness of your implementation.

- Follow style guidelines — include a class block comment with name, date, and that describes the class; include comments before each method; use inline comments as appropriate; make judicious use of whitespace, good naming convention, consistent indentation, etc.

Submitting:

1. First, you will need to package your work into a **tar** archive file for submitting (`tar` is required here so that you are required to learn how to use it). Let's assume that you have stored all necessary files in a directory called `lab2` (which may contain subdirectories). From the **parent** directory of your `lab2/` directory, execute the following command, replacing `netID` with your own netid:

   ```
   tar -czvf cmsc240_lab2_netID.tgz lab2/*
   ```

   This command will combine all the files and subdirectories in your `lab2/` directory into a single tar archive file named `cmsc240\_lab2\_netID.tgz`. (Have a look at the tar man-page about the `czvf` flags.)

   Henceforth, we will refer to such a file as a "gzipped tarball".

2. Verify the contents of your gzipped tarball. First, verify that it is of reasonable size (in bytes) and that it is identified as a "gzip compressed data" file:

   ```
   ls -l
   file cmsc240_lab2_netID.tgz
   ```

   The first command gives you a long listing of files in the directory, which should provide the size, in bytes, of all of the files in the directory (well, not all – files that begin with a dot, like `.bash_profile`, and some other special files, won't show up). The `file` command provides the file type. In this case it should say something like

   ```
   testing.tar: POSIX tar archive
   ```

   Next, check that the contents are what you expect:

   ```
   tar -tzvf cmsc240_lab2_netID.tgz
   ```

   This should give you a listing of the folders and files containedor in your tarball.

3. Attach the gzipped tarball to an email to the appropriate assignment email address (as usual) to submit.

Naming:

As always, because all students will be submitting their lab to a single Box folder, the files can't all be called "Lab2.tar" or the like. Instead, they must have some indication of who completed the submission. Naming is also important because I sometimes use software to grade submissions, and my code expects specific naming conventions. For this lab, your submission **MUST** be named `cmsc240_lab2_netID.tar`, where, of course the `netID` part is your netID.

Submission:

The high level picture is that to submit any labs/project in this course, you send an email to a special email address with your **single** submission file attached. This has the effect of placing your submission in the appropriate Box folder. If your submission requires more than one file, you should create a gzipped tarball as mentioned above.

The the email address for this lab is `lab2.tueuo0d5kp2ib0i3@u.box.com`. Thus to submit this lab, you should attach your gzipped tarball to an email sent to the email address `lab2.tueuo0d5kp2ib0i3@u.box.com`.