School Computing, Faculty of Engineering
Universiti Teknologi Malaysia

Programming Technique II
Semester 1, 2021/2022

Exercise 3
String Manipulations

Overview

- This exercise is to be conducted **outside of the class.**
- You will be adopting a **Pair Programming** strategy in doing this exercise.
  What is pair programming? (https://youtu.be/oBraLLybGDA)
- You can maintain your partner from the previous exercise or change to a new one.
- You and your partner will be coding collaboratively online using VS Code and Live Share.
  Using Live Share for online collaborative coding (https://youtu.be/s9hfONtUcR8)
- You will communicate to each other using Google Meet, Webex or any online meeting tool.
- You will record your pair programming session.

Pair Programming and Collaborative Coding

- Pick any time worth **TWO (2) hours** (maximum) within the given date to conduct the pair programing session with your partner.
- You may also split your pair programming into several sub-sessions provided the total time is still within 2 hours.
- Log the date and time for every pair programming session conducted. Write them in the program source code.
- Record the meeting about your pair programming session. If you do your programming in multiple sessions, record all of them. You do not have to edit the video.
- Code submissions without the video at all or the video was too short, will be declined.

Notes:

- You are advised to explore the exercise on your own first before doing the pair programming session with your partner. This should make yourself be more prepared for the pair programming session.

How To Record the Session

- You can choose Google meet as your online meeting tool and use the feature **“record meeting”** to record your pair programming session.
- Note that a free personal google account does not support the recording feature.
- Try using your student or graduate account from UTM to be able to access the “record meeting” feature.
- Alternatively, you can record your pair programming session locally using software like OBS, PowerPoint, etc. https://elearning.utm.my/21221/mod/page/view.php?id=26194

About the Video

- The video must show that you are coding, communicating, and collaborating with your partner. In this regard, **speak only in English**.
- In the video you should show your VS Code and the output (console).
- You can record the session in a single or multiple videos. If you use multiple videos, put them in a folder, and submit only the folder’s link.
- Set the video file (or folder) permissions so that “Anyone can view”.
- Make sure the video is available until the end of the semester.
- Submit the raw videos, i.e., you don’t have to do post-editing.

Notes:

- Please make the font-size of your VS Code a little bit larger so that it easy for me to see your code in the video. You can do this by pressing the key **Ctrl** and **+** in VS Code.

Plagiarism Warning

You may discuss with others and refer to any resources. However, any kind of plagiarism will lead to your submission being dismissed. No appeal will be entertained at all.

Late Submission and Penalties

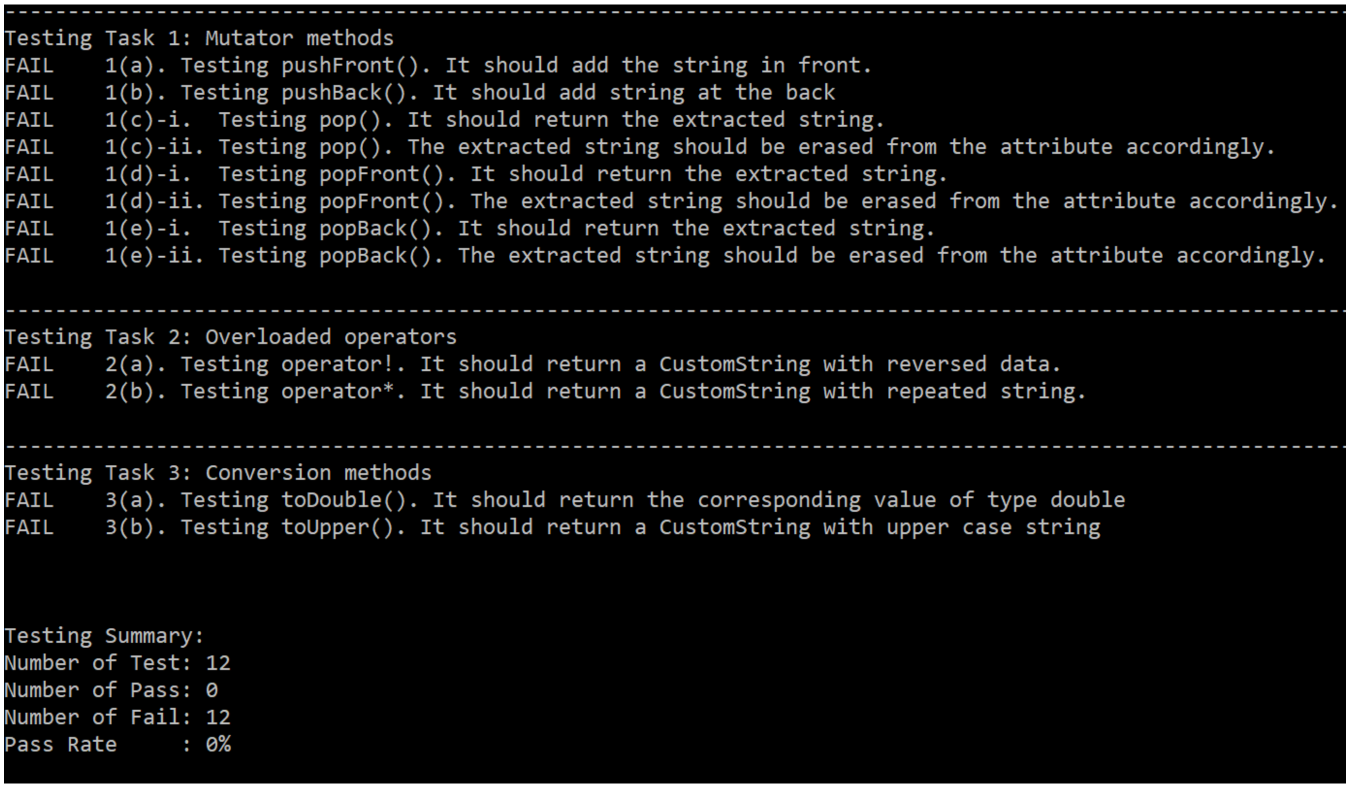
- The submission must be done via eLearning. Other than (such as telegram, email, google drive, etc.), it will not be entertained at all.
- Programs that CANNOT COMPILE will get a 50% penalty.
- Programs that are submitted late will get a 10% penalty for every 10 minutes.

Question

In this exercise, you will be writing a C++ program that defines a custom string class .You will write the program into separate files.

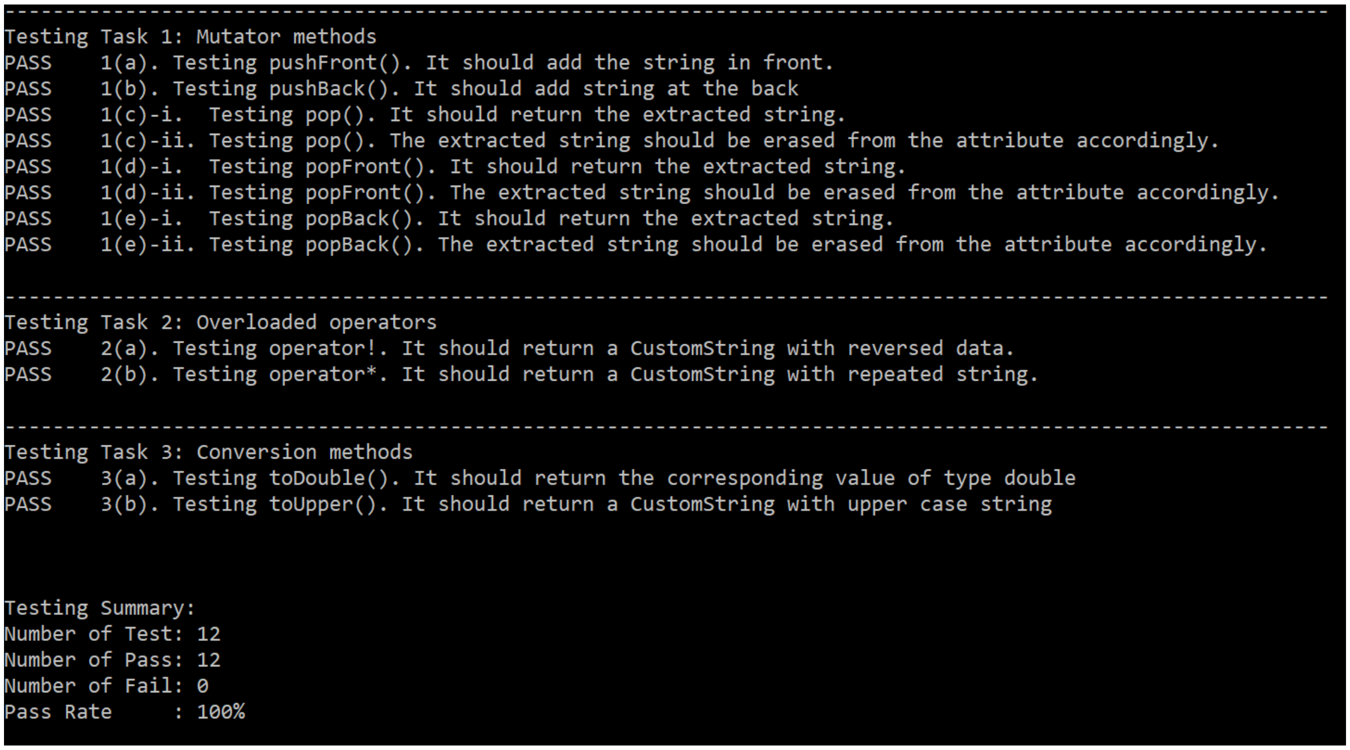
You are provided with a codebase containing a class named `CustomString` consisting of an attribute of type `string`, named `data`, several methods, and operators. The class declaration has already been given fully in the specification file, `custom_string.hpp`. Your job in this exercise, is to complete the class implementation.

The main file, `main.cpp` has also been given fully. The purpose of the main file is to perform automatic testing on the tasks you have accomplished. When you run the program with the original codebase, you should notice that all tests FAIL (as shown in Figure 1). Your job is to turn the tests to "PASS" (like in Figure 2) by accomplishing the tasks in this exercise.



```
-------------------------------------------------------------------------------------------------------------
Testing Task 1: Mutator methods
FAIL    1(a). Testing pushFront(). It should add the string in front.
FAIL    1(b). Testing pushBack(). It should add string at the back
FAIL    1(c)-i.  Testing pop(). It should return the extracted string.
FAIL    1(c)-ii. Testing pop(). The extracted string should be erased from the attribute accordingly.
FAIL    1(d)-i.  Testing popFront(). It should return the extracted string.
FAIL    1(d)-ii. Testing popFront(). The extracted string should be erased from the attribute accordingly.
FAIL    1(e)-i.  Testing popBack(). It should return the extracted string.
FAIL    1(e)-ii. Testing popBack(). The extracted string should be erased from the attribute accordingly.

-------------------------------------------------------------------------------------------------------------
Testing Task 2: Overloaded operators
FAIL    2(a). Testing operator!. It should return a CustomString with reversed data.
FAIL    2(b). Testing operator*. It should return a CustomString with repeated string.

-------------------------------------------------------------------------------------------------------------
Testing Task 3: Conversion methods
FAIL    3(a). Testing toDouble(). It should return the corresponding value of type double
FAIL    3(b). Testing toUpper(). It should return a CustomString with upper case string



Testing Summary:
Number of Test: 12
Number of Pass: 0
Number of Fail: 12
Pass Rate     : 0%
```

Figure 1: Run with the original codebase



```
-----------------------------------------------------------------------------------------------------------
Testing Task 1: Mutator methods
PASS    1(a). Testing pushFront(). It should add the string in front.
PASS    1(b). Testing pushBack(). It should add string at the back
PASS    1(c)-i.  Testing pop(). It should return the extracted string.
PASS    1(c)-ii. Testing pop(). The extracted string should be erased from the attribute accordingly.
PASS    1(d)-i.  Testing popFront(). It should return the extracted string.
PASS    1(d)-ii. Testing popFront(). The extracted string should be erased from the attribute accordingly.
PASS    1(e)-i.  Testing popBack(). It should return the extracted string.
PASS    1(e)-ii. Testing popBack(). The extracted string should be erased from the attribute accordingly.

-----------------------------------------------------------------------------------------------------------
Testing Task 2: Overloaded operators
PASS    2(a). Testing operator!. It should return a CustomString with reversed data.
PASS    2(b). Testing operator*. It should return a CustomString with repeated string.

-----------------------------------------------------------------------------------------------------------
Testing Task 3: Conversion methods
PASS    3(a). Testing toDouble(). It should return the corresponding value of type double
PASS    3(b). Testing toUpper(). It should return a CustomString with upper case string



Testing Summary:
Number of Test: 12
Number of Pass: 12
Number of Fail: 0
Pass Rate     : 100%
```

Figure 2: Expected Final Result

Modify only the class implementation file, **`custom_string.cpp`** to do the following tasks.

1. Complete the implementations of the mutator methods:

   a. `void pushFront(const string &s);`

   This method merges the string **`s`** to the object's attribute **`data`** in front. For example, if we have an object `cs`*:* `CustomString cs("World");`

   then this operation*:* `cs.pushFront("Hello ");`

   will modify the **`data`** of **`cs`** to `"Hello World"`.

   b. `void pushBack(const string &s);`

   This method merges the string **`s`** to the object's attribute **`data`** at the end. For example, if we have an object `cs`*:* `CustomString cs("C++ ");`

   then this operation*:* `cs.pushBack("Programming");`

   will modify the **`data`** of **`cs`** to `"C++ Programming"`.

c. **`string pop(int index, int count);`**

This method extracts and returns a substring from the object's attribute **`data`** starting from the **`index`** and taking as many as **`count`** characters. The extracted substring will also be erased from the object's `data`. For example, if we have an object `str`*:*

`CustomString str("ABCDEF");`

then this operation*:* `string result = str.pop(2,3);`

will give the `result = "CDE"` and leaving the `str`'s `data` to `"ABF"`.

d. **`string popFront(int count);`**

This method extracts and returns a substring from the object's attribute **`data`** starting from the beginning and taking as many as **`count`** characters. The extracted substring will also be erased from the object's `data`. For example, if we have an object `str`*:*

`CustomString str("ABCDEF");`

then this operation*:* `string result = str.popFront(3);`

will give the `result = "ABC"` and leaving the `str`'s `data` to `"DEF"`.

You should implement this method by making use of the method `pop()` from 1(c).

e. **`string popBack(int count);`**

This method extracts and returns a substring from the object's attribute **`data`** at the end by taking as many as **`count`** characters. The extracted substring will also be erased from the object's `data`. For example, if we have an object `str`*:*

`CustomString str("ABCDEF");`

then this operation*:* `string result = str.popBack(2);`

will give the `result = "EF"` and leaving the `str`'s `data` to `"ABCD"`.

You should also implement this method by making use of the method `pop()` from 1(c).

2. Complete the implementations of the overloaded operators:

   a. `CustomString operator!() const;`

   This operator returns a new `CustomString` object with a reversed `data`. This method does not change at all the original data of the current object. For example, if we have an object `cs`*:* `CustomString cs("abcdef");`
   then this operation*:* `CustomString result = !cs;`
   will give the `result`'s `data = "fedcba"` and the `cs`'s `data` remains as `"abcdef"`.

   b. `CustomString operator*(int count) const;`

   This operator returns a new `CustomString` object with repeated string from the `data` of the current object. This method does not change at all the original data of the current object. For example, if we have an object `cs`*:* `CustomString cs("XYZ ");`
   then this operation*:* `CustomString result = cs * 4;`
   will give the `result`'s `data = "XYZ XYZ XYZ XYZ "` and the `cs`'s `data` remains as `"XYZ "`.

3. Complete the implementations of the conversion methods:

   a. `double toDouble() const;`

   This method returns the corresponding value the string in `double` type. For example, if we have an object `str`*:* `CustomString str("99.50");`
   then this operation*:* `double number = str.toDouble();`
   will give the `number = 99.50`.

b. `CustomString toUpper() const;`

This method returns a new `CustomString` object with capitalized string from the `data` of the current object. This method does not change at all the original data of the current object. For example, if we have an object `str`*:* `CustomString str("One Plus Two");` then this operation*:* `CustomString result = str.toUpper();` will give the `result`'s `data = "ONE PLUS TWO"` and the `str`'s `data` remains as `"One Plus Two"`.

Assessment

This exercise carries **3%** weightage for the final grade of this course. The breakdown weightage is as follows (out of 100 points):

Criteria	Points
The code	
1. Task 1 – Implementations of the mutator methods:	
a. `pushFront()`	10
b. `pushBack()`	10
c. `pop()`	10
d. `popFront()`	5
e. `popBack()`	5
2. Task 2 – Implementations of the overloaded operators:	
a. `operator !`	10
b. `operator *`	10
3. Task 3 – Implementations of the conversion methods:	
a. `toDouble()`	10
b. `toUpper()`	10
1. Pair Programming Session	
a. Video and overall	10
b. Active collaboration	5
c. Both members play both roles Driver and Navigator.	5

Submission

- Deadline: Tuesday, 21 December 2021, 11:59 PM
- Only one member from each pair needs to do the submission.
- Submission must be done on eLearning. Any other means such as email, telegram, google drive will not be accepted at all.
- You will need to submit TWO (2) items:
  a. Source code: submit only the implementation file for the class, i.e., **custom_string.cpp**.
  b. Video link. Submit the link via eLearning as well.

FAQs

1. **Who should be my partner?**

   Choose your own partner. You may maintain your partner from the previous exercise.

2. **Can I pair up with someone from a different section?**

   No.

3. **Can I do the exercise alone?**

   This is only allowed if the number of students in the class is imbalanced. You also need to ask for permission from the lecturer.

4. **What do we need to show in the video?**

   You should show that you are **doing pair programming** rather than explaining about your code. The video is not meant for presentation.

5. **Do we need to switch roles between Driver and Navigator?**

   Yes. Your video should show that you and your partner keep switching between these two roles. No one should be dominant or play only one role.

6. **What if I do this exercise alone? Do I still need to submit the video?**

   In case you got permission to do the exercise alone, you still need to submit the video. You show in the video your progress in doing the exercise. You need to talk about what you are currently coding.

7. **What if we do pair programming physically (face to face).**

   You and your partner should use only one computer and sit side-by-side. You do not have to open LiveShare and online meetings. You can record the video locally using software like OBS. Again, you still need to talk and discuss with your partner in the video. It is also recommended to turn on the web camera. Keep in mind that you keep following the SOP about COVID-19 when working in a face-to-face environment.