CSE 232 Spring 2016

Programming Project #10

Assignment Overview

We are going to work with classes that manipulate dynamic memory.

This assignment is worth 65 points (6.5% of the course grade) and must be **completed and turned in before 11:59 on Monday, April 18th .**

Background

When you type in a text editor, you usually have a line of text and a cursor on that line, something like the below:

```
Mary has ▌lamb
```

The ▌ is the location of the cursor. If you were to type in this situation you would basically be doing an **<u>insert</u>**. The insertion would happen **before** (to the left) the cursor. For example, if we typed an `a` and a `space`, the line of text would become

```
Mary has a ▌lamb
```

If we wanted to change "has" to "had", we would move the cursor to the **<u>left</u>** 3 spaces (usually with our arrow keys)

```
Mary has▌ a lamb
```

Then we would then do a backspace, which is a **<u>delete</u>** of the `s` before (to the left) of the cursor

```
Mary ha▌ a lamb
```

Then **<u>insert</u>** the new letter `d` to fix the sentence.

```
Mary had▌ a lamb
```

We could then move three spaces to the **<u>right</u>** (again with our arrow keys)

```
Mary had a ▌lamb
```

And **<u>insert</u>** the characters to fix the sentence

```
Mary had a little ▌lamb
```

The four operations bold/underlined above: **<u>insert</u>**, **<u>delete</u>**, **<u>left</u>**, **<u>right</u>** are operations on a single line of text that would would like to implement. We are going to make a data structure that is specifically designed to deal with these four operations in an efficient manner.

Data Structure
The name of our data structure (and therefore our class) will be `TextBuffer`. The guts of a `TextBuffer` is a fixed array of characters. Let's look at how the work we did above match with this array.

Let's call our underlying array `buf_`, empty at first, with two special indicies. One is called `cursor_` and one is called `back_`. This is what `buf_` looks like at the beginning (empty). Note `buf_` is size 5

```
index: 0  1  2  3  4
buf_ :
cursor_ = 0,  back_=4
```

There is a ***gap*** between the `cursor_` and the `back_`. It is this gap that makes things easier. Lets look at the four operations:

Insert
We make a decision that `cursor_` is the index *where the next insert will go*. So `cursor_` is the index before/to-the-left of the cursor we saw in our editor example. To do an insert, we set `buf_[cursor_]` to the new character then add 1 to `cursor_`. Let's insert `'c'`:

```
index: 0  1  2  3  4
buf_:  c                →          c|
cursor_ = 1,  back_=4
```

Since `cursor_` increased by one, and the next insertion will happen after/to-the-right of `'c'`. Let's insert `'a'` and `'t'` to make `"cat"`

```
index: 0  1  2  3  4
buf_:  c  a  t          →          cat|
cursor_ = 3,  back_=4
```

Deletion
Deletion is very easy. To do a deletion we just move the cursor one to the left. Let's delete `'t'`

```
index: 0  1  2  3  4
buf_:  c  a  t           →          ca|
cursor_ = 2,  back_=4
```

Whoa, wait a second. Nothing changed! Well, something did. `cursor_` is now one less than it was. The next insertion will be at the `cursor_` location, overwriting the `'t'` at `buf_[2]`. We don't have to erase now. We will when we do an insertion. Nice! To remind us that `'t'` isn't really there we'll mark it with a color and bold font.

Left and Right
We haven't used the `back_` index yet. We need that for moves left and right. When we move the cursor we want to maintain the gap so we can do easy insert and delete (as above). Let's do 1 move to the left:

```
index: 0  1  2  3  4
buf_:  c  a  t     a      →         c|a
cursor_ = 1,  back_=3
```

As we did with cursor_, we decide that back_ is the index of where to put the next character. If the text editor cursor is in front of 'c', then everything behind/to-the-right should be on the other side of the gap so delete and insert remain easy. So when we move the cursor 1 left, the 'a' is moved to the end of the gap that is indicated by back_. Then cursor_ - 1 and back_ - 1 . cursor_ still indexes to the next insert, back_ indexes the next place for a move. Again, the left side 'a' is not erased, but it will be on the next insert. Again we'll mark it for convenience.

Let's insert the letter 'r'

```
index: 0  1  2  3  4
buf_:  c  r  t     a      →        cr|a
cursor_ = 2,  back_=3
```

And now lets move one right. If we move to the right, the letter behind/to-the-right the text cursor should move to the other side of the gap, so

```
index: 0  1  2  3  4
buf_:  c  r  a     a      →        cra|
cursor_ = 3,  back_=4
```

The letter 'a' indexed by back_ is again not "really there" but we don't bother to erase it since the next move will reset it. We mark it for convenience

Dynamic Growth.
Let's keep adding letters. We add 't'

```
index: 0  1  2  3  4
buf_:  c  r  a  t  a     →        crat|
cursor_ = 4,  back_=4
```

Let's move two left (we're on our way to make the word "create"). Again, that "not really there" 'a' is marked.

```
index: 0  1  2  3  4
buf_:  c  r  a  a  t     →        cr|at
cursor_ = 2,  back_=2
```

We try to add the 'e' but we cannot. We are full, no more gap (how can you tell!!!!) We need to make more room! Though you won't see any changes in the editor interface, we need to make more room in the array. The rule of thumb we will use is to double the size of the existing array. Given that buf_ is size 5, we grow it to 10 and fill from the original buf_. You need to:

- make a larger (x2) buffer
- copy the contents of buf_ to the new buffer
    - you have to mind the gap! Very important!
- update your size and capacity data
- update back_
- delete the old buf_

If you do it right, it should look like:

```
index: 0  1  2  3  4  5  6  7  8  9
buf_:  c  r  e                 a  t  →   cre|at
cursor_ = 3,  back_=7
```

Note the contents from `back_` and larger are at the ***end of the new buffer!!!***

Finally, two right, add the 'e'. Done.

```
index: 0  1  2  3  4  5  6  7  8  9
buf_:  c  r  e  a  t  e        a  t  →   create|
cursor_ = 6,  back_=9
```

Class Requirements

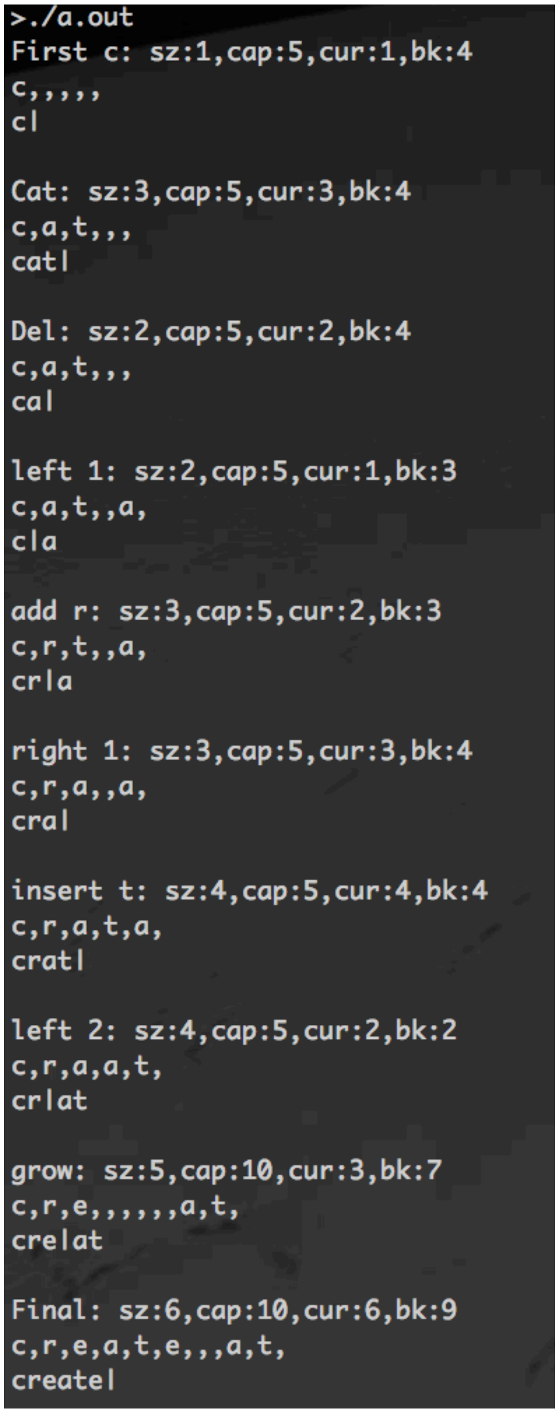
You are provided with the header file which should answer many questions. Here are some more details.

- `grow()` doubles the present size of `buf_` and does the appropriate copy. Should also delete any old memory.
- `swap()` swaps the values of two `TextBuffer`s. Note that inside of `swap`, you can use `std::swap` to swap each individual element. However, you ***must*** prefix `swap` inside of your swap with std::swap. See the class lecture notes
- `TextBuffer(size_t s=10)` default constructor parameter is the initial size of `buf_`
- copy constructor, destructor and operator= should be obvious
- `isfull()` returns `true` if there is no more gap. `false` otherwise
- `isempty()` returns `true` if there are no characters in `buf_`. `false` otherwise
- `insert(char c)` insert `c` as shown. If the `TextBuffer` `isfull()`, call `grow()` first then insert, otherwise just insert.
- `del()` If `cursor_` is all the way to the left, nothing and return false, otherwise delete as discussed and return true.
- `left()` If `cursor_` is all the way to the left, do nothing and return false, otherwise move left as discussed and return true.
- `right()` If `cursor_` is all the way to the right , do nothing and return false, otherwise move right as discussed and return true.
- `operator<<` My example below shows the output of a TextBuffer. Each call prints 3 lines:
  - first line, size, capacity, cursor value and back value
  - second line, the contents of buf_ (which you really need for debugging anyway so…)
  - third line, what the user might see as shown in the example. The `'|'` is used to represent the cursor location.

Deliverables

You are provided with class-10.h and main-10.cpp You must use handin to turn in the file: class-10.cpp. This is your source code solution; **be sure to include your section, the date, the project number** and comments describing your code. Please be sure to use the specified file names, and save a copy of the files to your H drive as a backup.

Example:
Using the example `main-10.cpp`, this is what I got for project output (follows the example):



```
>./a.out
First c: sz:1,cap:5,cur:1,bk:4
c,,,,,
c|

Cat: sz:3,cap:5,cur:3,bk:4
c,a,t,,,
cat|

Del: sz:2,cap:5,cur:2,bk:4
c,a,t,,,
ca|

left 1: sz:2,cap:5,cur:1,bk:3
c,a,t,,a,
c|a

add r: sz:3,cap:5,cur:2,bk:3
c,r,t,,a,
cr|a

right 1: sz:3,cap:5,cur:3,bk:4
c,r,a,,a,
cra|

insert t: sz:4,cap:5,cur:4,bk:4
c,r,a,t,a,
crat|

left 2: sz:4,cap:5,cur:2,bk:2
c,r,a,a,t,
cr|at

grow: sz:5,cap:10,cur:3,bk:7
c,r,e,,,,,,a,t,
cre|at

Final: sz:6,cap:10,cur:6,bk:9
c,r,e,a,t,e,,,a,t,
create|
```