CS 106L Winter 2020

Assignment 3: Gap Buffer

Assignment created by Avery Wang
Special thanks to Anna Zeng for brainstorming and testing!
Due date: Mar. 6, 2020, at 11:59 pm on Paperless.
See update log for minor updates. We'll send an email if there are major updates.

Update Log: (thank you to students who emailed us about these issues)
- updated due date is Friday at 11:59 pm
- removed the summaries at the end of each section. Makes the doc shorter.
- *removed to_external_index function, should not impact assignment as it shouldn't be called at all.*
- *clarified number of test cases - 42 official test cases + 2 init/clean up + 1 warning + 3 optional. We will only grade the 42 official test cases.*
- ***[Important] revised test cases so you can complete the assignment sequentially, and the corresponding test cases pass. See Piazza post.***
- *revised TestCase1F - incorrect behavior for delete_at_cursor when index is 0, should be a no-op.*
- *clarified number of lines for initializer list constructor.*

Part 1: Basics of Gap Buffers

Text editors are an important tool that computer scientists use. There is a constant debate over what the best text editor is, and Qt Creator is never one of the contenders.

One of the basic functions of a text editor is to store files while they are being edited. You can imagine a user moving the cursor around the file, and inserting and removing characters at the position of the cursor.

One way to store the characters in a file is to use a vector. We can simply store all the characters contiguously, and insert or remove characters at the index of the cursor, which we separately track (represented by the ^).

I	t	o	E	n		T	e	a							
									^						

cursor = 9

What if we wanted to edit this text so it reads "Ito-En Green Tea" (with the hyphen)? We would have to insert the character '-' at index 3. As you learned in CS 106B, the vector internally shifts all the characters after the '-' (the "En Tea") back, so the buffer reads:

I	t	o	-	E	n		T	e	a						
				^											

cursor = 4

This might be fine for a small file like this one, but imagine if we were editing a big book, like *Infinite Jest* by David Foster Wallace? The book has 543,709 words, or about 3 million characters. Trying to add or remove a single character in a vector requires moving 3 million characters by one index. In terms of Big-Oh, we represent the runtime as O(L), where L is the length of our text. We showed during lecture that being an editor of this text is extremely unpleasant.

Can we do better?

Seriously, try experimenting, and see if you can come up with something better.
This is how novel ideas (and startups) are born!

Memory Layout of a Gap Buffer

One observation is that to add or remove at a certain index, we have to move the cursor to that index. For a vector, moving the cursor is extremely fast (since it's just an index), but inserting at that index is slow. In a text editor, we will be inserting or removing characters much more frequently than jumping the cursor around. Let's do just a little more work moving the cursor around, to make insertion or removing faster.

A gap buffer is as simple as it sounds. It's a buffer (i.e. array), with a gap. This gap immediately follows the cursor's location. In the example above, once we've moved the cursor to index 3, our gap buffer looks like the diagram below. For simplicity, I've denoted the gap with asterisks (*), but in reality we don't care what the gap contains. We'll keep track of the cursor separately.

Warning: there are two indices here. We have the array index (labeled a), which is an index of the array, versus the external string index (labeled s), which is the index in the string that the user sees (since the user doesn't actually see the gap on her text editor). For example, in the diagram below, the character 'E' is stored at index 10a (since it is in index 10 in the array), but we also say it is at index 3s, since in the text editor it appears in index 3.

what the user sees = "Ito|En Tea", cursor_s = 3s

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
I	t	o	*	*	*	*	*	*	*	E	n		T	e	a
			^												

cursor_a = 3a
gap_size = 7

Insert and Delete at Cursor

Adding a character is fairly simple. Simply take up one of the spaces used by the gap. Let's say we wanted to move the cursor from index 3s to 4s. This takes constant time, O(1).

what the user sees = "Ito-|En Tea", cursor_s = 4s

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
I	t	o	-	*	*	*	*	*	*	E	n		T	e	a
				^											

cursor_a = 4a
gap_size = 6

What tradeoff did we make to achieve such a fast insertion time? Moving the cursor is slightly more difficult, because we need to shift the gap by moving characters around.

As a final example, suppose we start with the following, and press the backspace key three times. Here is what is happening in our gap buffer:

Start

what the user sees = "Ito-En Tea|", cursor_s = 10s

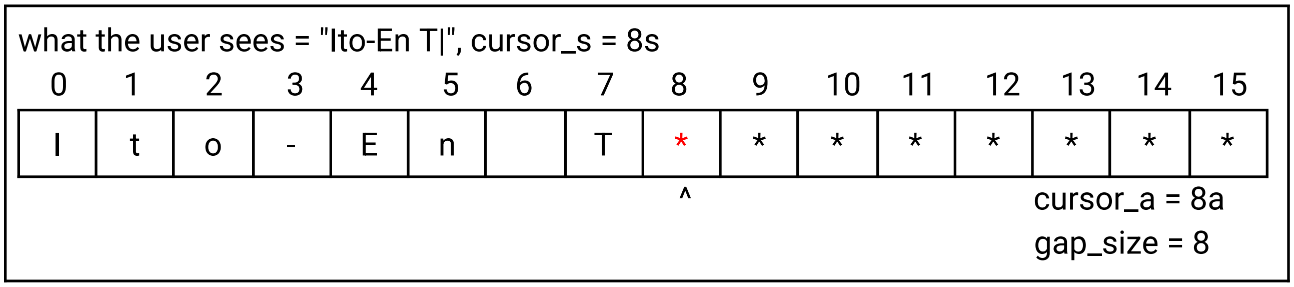
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
I	t	o	-	E	n		T	e	a	*	*	*	*	*	*
										^					

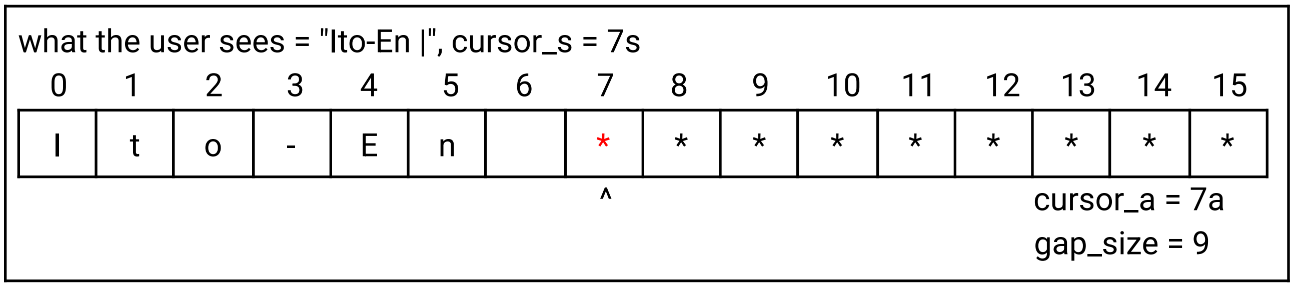
cursor_a = 10a
gap_size = 6

what the user sees = "Ito-En Te|", cursor_s = 9s

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
I	t	o	-	E	n		T	e	*	*	*	*	*	*	*
									^						

cursor_a = 9a
gap_size = 7


what the user sees = "Ito-En T|", cursor_s = 8s
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
I t o - E n T * * * * * * * *
^
cursor_a = 8a
gap_size = 8


what the user sees = "Ito-En |", cursor_s = 7s
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
I t o - E n * * * * * * * * *
^
cursor_a = 7a
gap_size = 9

Move and Reserve

The original version of the handout had move and reserve described, but since you won't need to implement these, we decided to make this handout a bit shorter. Feel free to take a peek at the code and figure out what's happening!

Note: it turns out that there is a memory leak inside reserve. We described how to fix it if solves any issues.

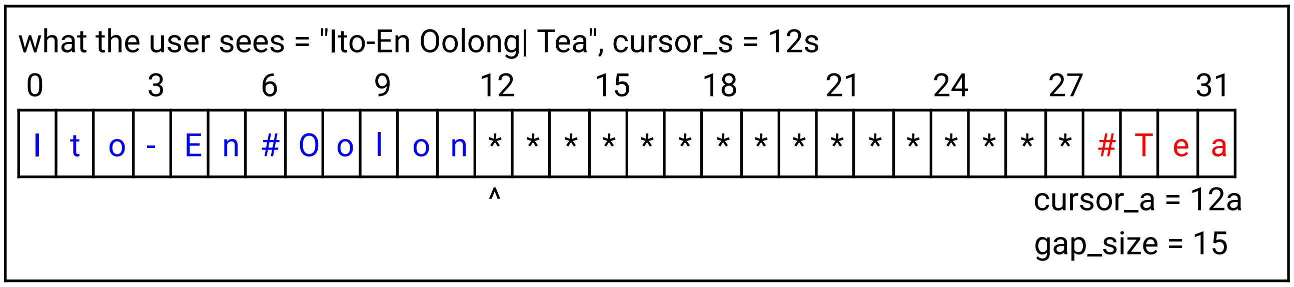
```
// 2. turns out the starter code had a massive memory leak.
//        in the reserve() function, add the following lines
//        _buffer_size = new_size;         (unchanged)
//        delete [] _elems;                (inserted)
//        _elems = std::move(new_elems);   (unchanged)
```

Element Random Access and Edit

What if we needed to access an arbitrary element (say, at index i)? This is useful if the editor needs to display everything on the display. Could we edit (though not remove or insert) any arbitrary element? This is definitely less useful - let me know if you think of a real example why an editor might want to do that?

In either case, since we don't need to shift any elements with element random access and editing, we can simply return a reference to an element by indexing into our array.

Be careful though! Accessing an element that is after the cursor requires some conversion between the user's index (as seen in the string) versus the array's indexing. For example, in the example below, if the user asks for index 14 (in the string), we need to know that this corresponds to the character at index 29 (the 'T').


what the user sees = "Ito-En Oolong| Tea", cursor_s = 12s
0
3
6
9
12
15
18
21
24
27
31
I t o - E n # O o l o n * * * * * * * * * * * * * * * * # T e a
^
cursor_a = 12a
gap_size = 15

We've provided a few helper functions called to_array_index which can help you convert between the two indices. Take a look at the functions, figure out what they are calculating, and use them!

Method	Lines	Description	Runtime

`GapBuffer();`
 	
`(Default constructor)`
	5	Creates an empty GapBuffer with the default size, and cursor at index 0.
		
		Hint: use a member initializer list.
			`O(1)`

`GapBuffer(size_type count, const char& ch = char());`
 	
`(Fill constructor)`
	6	Creates a GapBuffer that starts with *count* occurrences of *ch* (which has a default value of ch()). Cursor should be at the end of all of the characters, and the size of the buffer should be 2*count.	`O(S)`
			`s = len of string`

`void insert_at_cursor (const_reference element);`	7	insert given element at the current location of the cursor. If necessary, resizes the array by factor of 2.	`O(1)`
			`amortized if resize, O(N)`

`void delete_at_cursor();`	4	deletes element at current location of the cursor. *if cursor is at index 0, this is a no-op.*	`O(1)`
`reference get_at_cursor();`	5-6	returns the element immediately after the cursor. *if cursor is at index _logical_size, throw an exception.*	`O(1)`
`reference at(size_type pos);`	5-6	retrieves the element stored at external index pos, *performs bounds checking, throws an exception if out of bounds.*	`O(1)`

`void move_cursor(int delta);`	given	moves cursor left (if delta is negative) or right (if delta is positive)	`O(D)`
			`D = |delta|`

`void reserve (size_type new_size);`	given	if new_size is less than current size, do nothing,, otherwise, resize GapBuffer internal array to new_size	`O(N)`
			
			`N = num of elems`

`size_type size() const;`	1	returns the external size of the GapBuffer	`O(1)`
`bool empty() const;`	1	returns true if the GapBuffer is empty, false otherwise	`O(1)`

`void debug() const;`	given	prints the contents of your GapBuffer.	`O(A)`
			`A = size of array`

Part 2-3: Operators and Const Correctness

Part 2
In this part, you should make your GapBuffer class const correct. For methods which return a reference, you should return both a const and non-const version of these methods. Recall the static_cast/const_cast trick we learned during lecture to avoid repeated code.

Part 3
You should support the index operator, stream insertion operator, and all comparison operators for your Gap Buffer class. Try to think about why an operator should be overloaded as a member vs. a non-member, and about what the parameters, return value, and const-ness of the types are.

The comparison operators exist so that you can put a GapBuffer into a std::set. We consider two GapBuffers to be equal if all of their elements match (regardless of cursor position. You may find std::equal to be helpful (though you'll have to implement iterators in part 5 first). We consider GapBuffer A to be less than GapBuffer B by checking the first mismatching element (a, b), and checking if a < b. We also consider A to be less than B if the array of A forms a prefix of B. You may find std::lexicographical_compare to be helpful.

There are many reasonable ways to print a GapBuffer to cout. We'll print with the following rules:

- If there are no elements, print {^}
- If there are elements, then place a ^ in front of the element that will be removed by a call to delete_at_cursor, but before the opening brace or comma. If the cursor is at the end, place a ^ after the element, but before the closing braces.

Examples:
```
{1, 2, ^3, 4, 5}                     Contents: 12|345
{^1, 2, 3, 4, 5}                     Contents: |12345
{1, 2, 3, 4, 5^}                     Contents: 12345|
{^}                                  Contents: |
{^1}                                 Contents: |1
{1^}                                 Contents: 1|
{1, 2^}                              Contents: 12|
{1, ^2}                              Contents: 1|2
```

Method	Lines	Description	Runtime
`const_reference get_at_cursor() const;`	1-3	const version of get_at_cursor *Hint: use static_cast/const_cast trick in non-const version.*	`O(1)`
`const_reference at (size_type pos) const;`	1-3	const version of at *Hint: use static_cast/const_cast trick in non-const version.*	`O(1)`
`reference operator[] (size_type pos);`	1	non-const version of operator[ ] as a member. *Hint: use static_cast/const_cast trick.*	`O(1)`
`const_reference operator[] (size_type pos) const;`	1-3	const version of operator [ ] as a member.	`O(1)`
`std::ostream& operator<<(std::ostream& os, const GapBuffer& buf);`	18	non-member, **non-friend** overload of << for ostreams, prints string representation of buf to os (see note)	`O(N)`

`bool operator==(left, right)`
`bool operator!=(left, right)`
 	
`(Equality operators)`
	1-3 each	non-member, **non-friend** operator overload, returns whether or not all the elements are equal and the GapBuffer is of the same size. Ignore the position of the cursor when comparing.
		
		HInt: std::equal
			`O(N)`

`bool operator<(left, right)`
`bool operator>(left, right)`
`bool operator<=(left, right)`
`bool operator>=(left, right)`
 	
`(Comparison operators)`
	1-3 each	non-member, **non-friend** operator overload, returns the comparison of the first mismatching element between left and right. If one GapBuffer is a prefix of another, return the shorter one. Ignore the position of the cursor when comparing.
		
		Hint: std::lexicographical_compare
			`O(N)`

Part 4: Template Classes

Turn your GapBuffer into a class that supports an arbitrary type T. Your current class behaves like a GapBuffer<char>. Modify the class to become GapBuffer<T>.

- Step 1: download the new starter code, labelled "GapBuffer Full" - the template class is defined for you already.
- Step 2: move all your code from Parts 1-3 to the new starter code. We've given you the the function headers.

Part 5: Iterator Classes

Can we equip our class with iterators, so that we can use for-each loops, and even use the STL algorithms on our GapBuffer?

We will add support for random access iterators for the GapBuffer class, by implementing an iterator inner class. Here are the steps:

1. Declare an inner class inside GapBuffer called iterator, which is mutually friends with GapBuffer, and should inherit from std::iterator with the random access tag (done for you).
2. Add private members, such as a pointer to the GapBuffer object and an index of where the iterator is pointing right now. (done for you)
3. Add a constructor to the private section of the GapBuffer, which initializes the pointer to the GapBuffer object and the index (done for you)
4. **Overload a lot of operators to make the iterator a random access iterator.**
5. Add three methods begin(), end(), and cursor() to the GapBuffer class, which return an iterator to the begin, end, and cursor, respectively. (done for you)

In lecture, we discussed the design choices behind steps 1-3 and 5. The syntax for these steps is tricky and confusing, so we've provided it for you.

Recall the difference between the prefix and postfix operators for iterators. Both end up inc/decrementing the index. However, the prefix operator returns a reference to the inc/decremented iterator (itself), while the postfix operator returns a copy of the old iterator, while the new iterator (itself) has already been inc/decremented.

The compiler distinguishes the two by giving the postfix iterator a completely unused int parameter. There's no real reason behind this, and we usually don't even name that parameter.

Functions in GapBuffer class:

Method	Lines	Description	Efficiency

`iterator begin();`
`iterator end();`
`iterator cursor();`
	given	returns an iterator at the respective position	`O(1)`

Methods in GapBuffer::iterator class. Member are in green, non-member friends are in red.

Method	Lines	Description	Efficiency
`reference& operator*();`	1	dereference operator, retrieves element stored at external index	`O(1)`

`iterator& operator++();`
`iterator& operator--();`
	2 each	prefix increment/decrement operator, moves the iterator forward/backward by one element, and returns reference to itself	`O(1)`

`iterator operator++(int);`
`iterator operator--(int);`
	3 each	postfix increment/decrement operator, moves the iterator forward/backward by one element, but **returns a copy of the iterator before the increment/decrement**	`O(1)`

`iterator operator+ (const iterator& lhs, size_t diff);`
 	
`iterator operator+ (size_t diff, const iterator& rhs);`
 	
`iterator operator- (const iterator& lhs, size_t diff);`
	3 each
	
	given
		outputs a **new** iterator that is the current iterator inc/decremented by diff positions.
		
		last non-member allows cases when the user writes "3 + iter" instead of "iter + 3".
		
		Hint: write these operators in terms of += and -=, which are written for you.
			`O(1)`

`bool operator==(left, right)`
`bool operator!=(left, right)`
`bool operator<(left, right)`
`bool operator>(left, right)`
`bool operator<=(left, right)`
`bool operator>=(left, right)`
	given	compares the external indices pointed to by the two iterators.
		
		these are implemented as non-members, consistent with the principles we discussed in lecture.
			`O(1)`

`difference_type operator- (lhs, rhs) const;`	given	finds the difference in positions of the iterator to itself.	`O(1)`

`iterator& operator+=(size_t diff;`
 	
`iterator& operator-=(size_t diff);`
	given	inc/decrements iterator by diff positions, and returns reference to itself.	`O(1)`

`reference& operator[](size_t index);`	given	dereferences element at index (current + index)	`O(1)`

Part 6: Special Member Functions

As discussed during lecture, our GapBuffer does not correctly copy when we attempt to copy it using the assignment operator, and it also leaks memory. The reason is that we are copying the pointer to _elems, resulting in the original GapBuffer and the copy to point to the same array. To solve this, we will implement the special member functions.

Following the rule of three, you should also implement the copy assignment operator. The main difference is that the copy assignment operator needs to free the memory associated with the existing object about to be overwritten, and needs to check for self-assignment.

Method	Lines	Description	Efficiency
`GapBuffer();`	1	you already wrote this, now add a member initialization list	`O(1)`

`GapBuffer(size_type count, const value_type& val = value_type());`
 	
 `(Fill constructor)`
	4	you already wrote this, now add a member initialization list	`O(N)`

`GapBuffer(std::initailizer_list<T> init);`
 	
 `(Initializer list constructor)`
	2-6*	creates a GapBuffer with elements from initializer list init.
		
		 * we used 2 lines by calling the fill constructor. Feel free to implement it however you'd like.
			`O(N)`

`GapBuffer(const GapBuffer& other);`
 	
 `(Copy constructor)`
	4	copy constructor: creates an object that is a copy of other.	`O(N)`

`GapBuffer& operator=(const GapBuffer& other);`
 	
 `(Copy assignment)`
	8	copy assignment: copies the data from other to this. Remember to free the data of this before copying.	`O(N)`

`~GapBuffer();`	2	frees resources held by GapBuffer	`O(1)`

Part 7: Move Semantics

We will now incorporate move semantics into our class. Recall that we are allowed to move data rather than copy when we know the data is a temporary variable. In other words, the variable passed in binds to an r-value reference. We will implement the following:

Method	Lines	Description	Efficiency
`GapBuffer(GapBuffer&& other);`	4	move constructor - moves the data from other, and uses it to construct a new GapBuffer. other remains in an valid but undetermined state.	`O(N)`
`GapBuffer& operator=(GapBuffer&& rhs);`	7	move assignment - frees the data in this, moves the data from other to this. rhs remains in an valid but undetermined state.	`O(N)`
`void insert_at_cursor (value_type&& element);`	7	moves the parameter passed in and places it directly into the array without making a copy of the element.	`O(1)`

Part 8: RAII and Smart Pointers [Optional]

Adapt your GapBuffer to be RAII-complaint. In other words, make sure that no matter where an exception is thrown, your class will not leak resources. Change your usage of pointers to be smart pointers, when appropriate.

Part 9: Emplacement [Optional]

Method	Lines	Description	Efficiency
`void emplace_at_cursor (Args&&... args)`	5	Inserts an element that is constructed in-place using the variable number of arguments. This is called a variadic template.	`O(N)`

Requirements for Receiving Credit

As stated in the CS 106L syllabus, all students are required to complete assignment 3. The requirement to pass assignment 3 satisfactorily is to pass all provided test cases, not including the extra credit test cases or the warning test case. We do not have hidden test cases that we will run your code against.

There is a piazza post describing an updated test harness (rearranges the test cases so that the test cases pass in order, and removes the RAII test case). Make sure you copy paste the new test harness .cpp file into your own file. No need to change your .h code.

To submit, upload your *GapBuffer.h* file to Paperless.

If you run into issues, please email us or visit us at office hours, and we will help you debug your code!

FAQ

- **Why am I getting an error about begin not being marked as const when trying to use std::copy on my iterators, like the following piece of code.**

```
void foo (const GapBuffer& bar) {
      std::copy(bar.begin(), bar.end(), begin()); // bar.begin() is not marked const
}
```

The reason you get this error is that your begin() function is not marked as const. Normally, we would define another begin and end function that returns a const_iterator, which we decided to not require you to implement.

There is a quick fix to this, and this is to use a const_cast to turn bar into a non-const reference.

```
void foo (const GapBuffer& bar) {
      auto& bar_nc = const_cast<GapBuffer&>(bar);
      std::copy(bar_nc.begin(), bar_nc.end(), begin()); // OK!
}
```

The reference in bar_nc is very important! If you don't declare it as a reference, this is either a copy or move operation. If you are doing this in your constructor, you have an infinite recursion on your constructor.

- **I'm getting warnings about the order of my member initializer list. What does that mean?**

You need to initialize the members in the same order you declared them in the class declaration. Feel free to change the order of the declaration if you feel that is more convenient for you.

- **I'm getting weird memory errors and unexpected crashes. Why?**

Possible reasons: order of member initializer list (see previous point), or you are walking into uninitialized memory. Very common mistake occurs when implementing operator<. When going through the two ranges, if one range is shorter than the other, don't stomp into uninitialized memory past the shorter range.

- **My part 1 tests are failing for some unknown reason. insert_at_cursor seems to work, but nothing seems to be inserted.**

The problem is that insert_at_cursor(1) is often times calling the r-value version of insert_at_cursor that you will implement in part 7, since 1 is an r-value. I recommend that you put the line "insert_at_cursor(element)" inside the r-value version right now, and remove that once you are ready to implement that part.