CS 202 - Extra Credit

Mr. Piotrowski

April 2022

1 Introduction

This assignment is designed to help you practice using templates. Your job is to develop a simple template Dynamic Array class, similar to a vector which is something we have used in past assignments. As the semester comes to a close with a few assignments left, there will be no special tricks here. Just make the class and make sure it works with the main.cpp file.

2 DynamicArray

```
template<typename Type>
class dynamicArray
```

	MAS	member	const	static	type	name	params
	private	variable	n	n	Type*	arr	N/A
	private	variable	n	n	unsigned int	currentSize	N/A
optional	private	variable	n	n	unsigned int	maxSize	N/A
	public	constructor	N/A	N/A	N/A	dynamicArray	
	public	constructor	N/A	N/A	N/A	dynamicArray	Type*, unsigned int
	public	constructor	N/A	N/A	N/A	dynamicArray	const dynamicArray<Type>&
	public	destructor	N/A	N/A	N/A	~dynamicArray	N/A
	public	operator	n	n	void	operator=	const dynamicArray<Type>&
	public	operator	n	n	Type&	operator[]	unsigned int
	public	function	y	n	Type	at	unsigned int
	public	function	y	n	unsigned int	getSize	
	public	function	n	n	void	pushBack	Type
	public	function	n	n	void	pushFront	Type
	public	function	n	n	void	insert	Type, unsigned int
	public	function	n	n	void	clearArray	
	public	function	n	n	void	deleteAt	unsigned int
	public	function	n	n	void	sort	
		friend operator	N/A	N/A	ostream&	operator<<	ostream&, dynamicArray<Type>
		friend operator	N/A	N/A	istream&	operator>>	istream&, dynamicArray<Type>&

Develop the above DynamicArray class. I will let you code it the way you want. There is no implementation file for this assignment, only the header file. You can leave all function definitions within the class, or you can bring them outside of the class.

You will notice up above there is an optional variable. When it comes to dynamic arrays (arrays that are re-sizable), memory must be requested from the operating system during the run time. This class could be implemented such that it requests memory from the OS and returns memory to the OS as little as possible. We will call this a **time-efficient dynamicArray**. The same class could also be implemented where memory is requested and returned to the OS on every insert or delete operation. We will call this a **memory-efficient dynamicArray**.

2.1 Time-Efficient Array

In a time-efficient array we want to talk to the OS about memory needs as little as possible. To implement an array like this, we will almost always have an array that has extra memory. In other words, the array is over-sized and the elements of the array only fill it partially. For example, the array may have space for 100 elements, yet there is only 50 elements currently being stored.

This type of array will be very fast on insert and delete operations because as long as there is space, you will just have to insert or remove an element, and then potentially shift some elements around. Now this may sound easier, but you must always consider the case when this array runs out of memory. When that happens, it will need to take some time to get a larger array. So, every insert function (insert, pushBack, pushFront) will need to be equipped to allocate more memory if needed.

If you are going to implement this type of array, you will need the extra variable to keep track of the max size the array can currently hold. The other size variable keeps track of how many elements are currently stored there.

2.2 Memory-Efficient Array

This type of array will be slower than the one above. This type of array will only have as much memory as it needs. Meaning if an array has size N, there will be N elements stored there. There will be no extra memory space. So, for every insert and delete, the array needs to be resized. You do not need the optional variable for this implementation as the size of the array is always the same as the number of elements.

2.3 DynamicArray Class

This class needs to be a template. Consider naming the template type **"Type"** so you can reference the class UML up above.

2.3.1 The Default Constructor

This constructor should just initialize the size to be zero and the arr to be nullptr. If you are doing a time-efficient array, you may wish to allocate some initial memory here. **Do not allocate an initial array with size 10,000+**. main.cpp will test your class with about this many elements and I need to verify your resizing operations happen at least once. Starting out with a huge array is not appropriate for this assignment and I will deduct at least 30% if you choose this shortcut. I would say an initial size of 100 is reasonable.

2.3.2 The Parameter Constructor

This constructor should make a deep copy of the parameter array. The size is passed in so you know how large the parameter array is.

2.3.3 The Copy Constructor

This constructor should make a deep copy of the parameter dynamicArray. You can use the assignment operator trick if you want, as long as you first initialize the member variables.

2.3.4 The Destructor

Deallocate the array.

2.3.5 void clearArray()

This function needs to reset the array to be empty. That is, the array will have no elements. If doing a memory-efficient approach, you will have to deallocate the array and update the class variables. If doing a time-efficient array, you may just reset the currentSize to 0.

2.3.6 void operator=(const dynamicArray<Type>&)

This operator needs to make a deep copy of the parameter array. Do not forget to delete the old memory before overwriting for a new array.

2.3.7 unsigned int getSize() const

This getter function returns the size of the array - that is, the number of elements in the array.

2.3.8 void pushBack(Type)

This function will insert the **Type** parameter (template) into the back of the array. Regardless of your approach, this function will require you to resize the array if needed. To describe it in detail, you need to allocate a new array of a larger size, deep copy all the old elements, insert the new element, deallocate the old memory, and overwrite the pointer with the address to the new array. You will also have to increase the size of course.

If you are doing a time-efficient approach, space will run out in the array eventually. When this happens, you can increase the size by many spaces. For example, if the array was maxSize 100, you may allocate an array to be maxSize + 100 (*200*) now. This way the object only needs to do memory operations every 100 elements.

2.3.9 void pushFront(Type)

This function will insert the **Type** parameter (template) into the front of the array. Regardless of your approach, this function will require you to resize the array if needed. To describe it in detail, you need to allocate a new array of a larger size, deep copy all the old elements, insert the new element, deallocate the old memory, and overwrite the pointer with the address to the new array. You will also have to increase the size of course.

If you are doing a time-efficient approach, space will run out in the array eventually. When this happens, you can increase the size by many spaces. For

example, if the array was maxSize 100, you may allocate an array to be maxSize + 100 (*200*) now. This way the object only needs to do memory operations every 100 elements.

2.3.10 void insert(Type, unsigned int)

This function will insert the **Type** parameter (template) into the array that the index parameter. All old elements at the index (or greater) will need to be shifted right. You should verify the index being inserted to exists. For example, a size 0 array has no valid indexes to insert into. Similarly, a size 100 array cannot have an element inserted at index 101.

Regardless of your approach, this function will require you to resize the array if needed. To describe it in detail, you need to allocate a new array of a larger size, deep copy all the old elements, insert the new element, deallocate the old memory, and overwrite the pointer with the address to the new array. You will also have to increase the size of course.

If you are doing a time-efficient approach, space will run out in the array eventually. When this happens, you can increase the size by many spaces. For example, if the array was maxSize 100, you may allocate an array to be maxSize + 100 (*200*) now. This way the object only needs to do memory operations every 100 elements.

2.3.11 void deleteAt(unsigned int)

This function will delete the element located at the index. If the index is out of bounds, just do nothing and return. Depending on your implementation, it will require you to allocate an new array of a smaller size, deep copy all the old elements except for the deleted element, deallocate the old memory, and overwrite the pointer with the address to the new array. You will also have to decrease the size of course.

In a time-efficient approach, there is no reason to resize the array because the array is expected to be oversized. You may just have to shift elements left to overwrite the element being deleted.

2.3.12 Type &operator[](unsigned int)

Return the element in the array located at the index. It should return the reference, so please add the ampersand in the operator header. The index is the int parameter. This operator allows read-write access to individual elements. If the index is out of bounds, you can just return a default Type object, or call exit(1) which will exit the program with error code 1.

2.3.13 Type at(unsigned int) const

Return the element in the array located at the index. It should return a copy, so please do not use the ampersand in the function header. The index is the int parameter. This function allows read-only access to individual elements. If the index is out of bounds, you can just return a default Type object, or call exit(1) which will exit the program with error code 1.

2.3.14 friend ostream& operator<<(ostream&, dynamicArray<Type>)

Output the whole array to the ostream and then return the ostream. Make sure there are square brackets on the outside of the array and commas between each element. Example below:

```
[ 2 , 3 , 5 , 8 , 10 ]

[ ]

[ Patty , Willa , Phoebe ]
```

2.3.15 friend istream& operator>>(istream&, dynamicArray<Type>&)

This operator should read in each element to the array and then return the istream. That is, if the currentSize of the array is N, read in that many elements. You will be overwriting the old elements.

2.3.16 void sort()

This function should sort the array in ascending order.

3 Compiling Your Code

Compile your code using

```
g++ -std=c++11 -g main.cpp
```

Run your code using

```
./a.out
```

Use valgrind!

```
valgrind --leak-check=full ./a.out
```

4 Submission

Submit cpp file to CodeGrade.