Question 2b

When calling `at(index)` with index greater than the size of the array, we had the option of either growing the array to the index and then returning the default value or returning the default value, or something else entirely (such as return a predetermined “this was illegal” value). We decided to simply return the default value since it doesn’t use extra memory and was simple to implement, but acts similar to the result if we had actually grown the array like some vector implementations do.

Similarly with `assign(index, val)` with the index greater than the size of the array we could have either grown the array to that size or caught the exception and done nothing. We chose to grow the array to the index provided since the client programmer will expect that value to exist at that point (whether or not it was valid) and doing nothing (especially with a void signature and no way of telling the client that an error occurred) would be counter-intuitive. The client programmer would continue to work, thinking that the operation was successful and possibly counting on it for future operations.

Finally when `resize` causes a `bad_alloc` we follow the guidelines provided by the assignment specification and attempt more resizes, followed by throwing a custom error.

Question 3

We considered the problem of a webpage owner who wants to create a subscription service where the subscribed users have the advertisements removed from the site, but have the unsubscribed users still see the ads. In order to do this, we will implement the decorator pattern so that an ads decorator adds the advertisements and the lack of an ads decorator indicates no ads will appear.

Original UML

![UML Diagram](image-url)
Decorator UML

Don't need HasAds() operations -- the presence or absence of the AdsDecorator determines whether a User sees ads or not