Q2 b)

at(index): If the index given is not within the bounds of our array, we simply return defaultValue. This is meant to emulate the functionality of trying to retrieve a value which has not been set yet, and replaces the throwing of an std::out_of_range exception. This is the default value that a user would expect to find in a place they haven’t set, which is very likely with an index larger than the size of the array. Other options were to return -1 to signal an error, but this simply translates the notification of an error to a more primitive form.

assign(): Instead of checking if index is out of range and throwing an exception, we first see if we can resize the array to accommodate the new index. We use the FlexArray::resize method because we are only interested in resizing it to the size of (index + 1), not in some value between that and our current size. After resizing (if it’s needed), we then try to assign the value using our original FlexArray::assign method. Both of these are inside a try, catch block which swallows the exceptions that can come from either function. This reduces the amount of out_of_range exceptions that our assign() method throws, which could otherwise be easily avoided by simply resizing the array.

If we are unable to resize the array due to an incredibly large index, we simply finish gracefully without an error. This is preferred over affecting our array data based on an index which is not in the array.

resize(): Our resize method tries to resize the array, and if unsuccessful tries a smaller than requested two more times. If the operation is still unsuccessful we throw a bad_alloc_retry exception which we created, and which inherits from std::bad_alloc. Our exception class has a “size” variable which stores the most recent value for newSize, and overrides the “what()” method to display a message along with the size to the user.

Q3 a)

We wish to have a way to create an HTML expression composed of some plain text, and any combination (including duplicates) of the following HTML tags: <b><i><u><a></a><span></span></b>

Both the <a> and <span> require extra information, namely a value for the “href” and “style” parameters respectively.

We want a way to easily create proper HTML expressions from the client code using the above tags. Here is some sample output:

<b><u><i><span style='margin-left: 70px;'>Plain text here!!!</i></span></i></b>

The following two figures represent 1) a UML model of the problem, and 2) a UML model of the problem with the decorator design pattern applied.