CSCI-1200 Data Structures — Fall 2020
Lab 4 — `Vec` Implementation

Checkpoint 1 ***estimate: 30-45 minutes***

Like last week, every member of your lab group should have received an email this morning from the instructor. The email contains the code for Checkpoint 1. Each member of your lab study group received a different fragment of code with a food group theme.

IMPORTANT: Do not share the code fragment with your teammates yet!

For *PART ONE* of this checkpoint, each team member should **draw the memory diagram that is produced by that short fragment of code**. Review the memory diagramming examples from Lecture 4 & 5 and follow the same diagramming conventions.

Next, for *PART TWO*, share this diagram (but not the code!) with your teammates. If you've drawn the diagram on paper, take a photo with your cell phone camera. If you've used a digital painting/drawing tool, save the diagram as a and image or .pdf. Share the diagram with all of your teammates using email, WebEx screensharing, or another method of document sharing.

Now the team should work together as a group to *REVERSE ENGINEER* and **deduce a fragment of code that will produce each diagram**.

After the group has finished discussing the diagram and written out the deduced code fragment, the student who prepared the diagram can share the original code fragment and the group should compare the original and reverse-engineered code fragments and discuss any differences and evaluate the accuracy of the diagramming and reverse-engineering process.

To complete this checkpoint, after your group has discussed each of the diagrams (or after 45 minutes of lab time has passed, whichever is first), present both your individual drawings and the groups work to a TA or mentor.

Checkpoints 2 and 3 explore our implementation from lecture of the STL `vector` class. Please download:

`http://www.cs.rpi.edu/academics/courses/fall20/csci1200/labs/04_vec_implementation/vec.h`
`http://www.cs.rpi.edu/academics/courses/fall20/csci1200/labs/04_vec_implementation/test_vec.cpp`

Checkpoint 2 ***estimate: 10-25 minutes***

Write a templated non-member function named `remove_matching_elements` that takes in two arguments, a vector of type `Vec<T>` and an element of type `T`, and returns the number of elements that matched the argument and were successfully removed from the vector. The order of the other elements should stay the same. For example, if `v`, a `Vec<int>` object contains 6 elements: `11 22 33 11 55 22` and you call `remove_matching_elements(v,11)`, that call should return 2, and `v` should now contain: `22 33 55 22`.

Add several test cases to `test_vec.cpp` to show that the function works as expected. Think about the efficiency of your solution relative to the size of the vector, and the number of occurrences of the input element in the vector. Make sure that the function is not unnecessarily wasteful of CPU or memory resources.

NOTE: Your implementation should not use iterators or the `erase` *function that is part STL* `vector` *implementation. We will cover that function in lecture soon. Your implementation should not create a new* `Vec` *object or use an array or STL* `vector` *as a helper structure.*

To complete this checkpoint, show a TA your debugged solution for `remove_matching_elements` and be prepared to discuss the efficiency of the function.

Checkpoint 3

estimate: 20-40 minutes

Add a `print` member function to `Vec` to aid in debugging. (Note, neither `remove_matching_elements` nor `print` are not part of the STL standard for `vector`). You should print the current information stored in the variables `m_alloc`, `m_size`, and `m_data`. Use the `print` function to confirm your `remove_matching_elements` function is debugged. Also, write a test case that calls `push_back` many, many times (hint, use a for loop!) and observe how infrequently re-allocation of the `m_data` array is necessary.

To verify your code does not contain memory errors or memory leaks, use Valgrind and/or Dr. Memory on your local machine – see instructions on the course webpage: Memory Debugging. Also, submit your code to the homework server (in the practice space for lab 4), which is configured to run the memory debuggers for this exercise. To verify that you understand the output from Valgrind and/or Dr. Memory, temporarily add a simple bug into your implementation to cause a memory error or memory leak.

To complete this checkpoint, show a TA your tested & debugged program. Be prepared to demo and discuss the Valgrind and/or Dr. Memory output: with and without memory errors and memory leaks *AND* on your local machine and on the homework server.