Abraham Meza

Christos Papachristos

CS 202 – 1001

May 01, 2019

Project 10 Documentation

For this project you will create a templated Stack class, with an Array-based and a Node-based variant. A Stack is a Last In First Out (LIFO) data structure. A Stack exclusively inserts data at the top (**push**) and removes data from the top (**pop**) as well. The Stack’s **m\_top** data member is used to keep track of the current Stack size, and through that also infer the position of the last inserted element (the most recent one).

The following provided specifications refer to Stacks that work with DataType class objects, similarly to the previous project. For the this Project’s requirements, you will have to make the necessary modifications so that your ArrayStack and NodeStack and all their functionalities are generalized templated classes.

Continuing through Computer Science II (202), our instructor assigned us a project that implements all of our previous knowledge on test drivers and class implementation in order to create a templated Stack class. Instead of completing a fragment of code, students are to reference the given MakeFile in order to create an array-based and node-based Stack program. Based on the MakeFile, the instructor only wants students to create a single programming file that derives its functions from two designated header files. In doing so, the instructor wants students to implement their functions and methods within the skeleton classes for each orientation. Similar to previous projects, we are instructed to create a sample driver that checks all required specifications. The purpose of this project is to create stack classes that can be utilized universally in order to showcase the essential methods of push() and pop(), while also teaching students how to create templated classes. In doing so, students will be able to be able to modify their classes to a templated (good practice) and will be able to modify stacks accordingly to incorporate accurate information and efficient dynamic memory handling, if needed. As students continue to polish their knowledge and different implementations with classes, they are also being evaluated to be able to prove the functionality of their program through driver testing of their own creation.

For my design, I referred to the MakeFile structure, and the provided instructions in order to outline the header files needed for each Stack program. By referencing the previous lab and lectures, I was able to declare the necessary constructors, friend operators, and method in a templated form. Compared to our usual skeleton structures, the templated skeletons simply required to be restructured into a generic skeleton (replacing data types to specified class template, parameterized size, etc). Once I had outlined both header files, I began to implement my array-based class. Given the many parameters, I was able to structure the program without much difficulty. I began to define the constructors and initialized/assigned the given variables to their corresponding members. As an array environment, I had to manipulate the data through the utilization of *for* loops for access to specific elements. Once I had assigned the data accordingly, I realized that the main functionality of stacks is to incorporate array modification through the utilization of push() and pop() methods as to line up data. Through the realization, I implemented the methods within the *for* loops in the constructors to maintain the structure of the array environment throughout the entire program. Another thought that presented itself was the method in which dynamic memory would be handled within the array-based stack. I had realized that there was no need to dynamically allocate/deallocate memory in an array-based stack due to the fact that it was always going to be consistent and precise (max limit), thus the destructor had no expected implementation.

After working on the constructors, I moved onto the individual method and operators. Unlike the queue assignments and projects we’ve had in the past, the methods incorporated into the class skeleton revolved around the idea of array accessing in a “vertical” environment: i.e., top() manipulates pointers to point to the most recent data form that was created, etc. Even so, the implementation of the positioning methods was self-explanatory as it required the concept of array indexing in reverse order (top – bottom). From thereafter, I began to focus on the stack methods. The most important methods in the program are push() and pop(). After researching the concepts of push and pop within stacked data, I was able to implement them through the use of *if* statements to check the status of the array (if empty() / full()) in order to see if the value passed as an argument could be added to the stack. While push() adds an assigned value to the top of the array-based stack, pop() simply removes the top value and reassigns m\_top to the next most-recent element. Since stacks depend on the order from top – bottom, I simply had to manipulate m\_top in order to add/remove data. After implementing the crucial methods, I continued with the helper methods that would simply check the size (m\_size) of array in order to perform conditional statements for other methods. Once I completed the methods, I began to work on the operators. The only operator that was asked for was the insertion operator overload. Based on the instructions, the insertion operator had to be connected to the serialize() method that displayed the format in which the data had to be printed. Similar to queues, stacks can be seen as a one dimensional array; through the utilization of *for* loops (array indexing), I accessed the element data and printed it to the screen accordingly.

After concluding and testing the array-based stack, I began to construct the node-based implementation. Though both stacks were to execute identical functionalities, their programming utilized different manipulations. While array-based lists utilized the idea of array indexing, node lists utilize pointer manipulation to access/modify specific data. Through my knowledge of pointer manipulation, I simply referenced my created array-based queue and modified it to be compatible with pointer arithmetic. By creating several pointer variables and objects that would track and transfer data, I was able to successfully implement the same methods. However, I quickly realized that object pointer declaration was different in a templated class. Instead of calling a direct class, I had to modify it to work with the generic template by referencing the parametrized declaration of a templated class (Node <T> instead of NodeStack).

As a result of using pointer arithmetic (arrow operators, dot operators, etc.), I could simply point to the next object. By accessing the data of the pointed object, I could manipulate it accordingly without much trouble. Through the utilization of new pointer declarations and counting variables, I was able to modify and control the data and dynamic memory. Since the node-based queue could have infinite members, I had to manage the dynamic memory throughout the program. I would simply delete old data and reallocated the memory of the node elements. Similar to the array-based stack, the helper methods simply tracked the size of the node through the utilization of temp variables and loops. All other methods utilized *while* loops to scan through the elements and used arrow operators to navigate through the positioning (m\_top: top – bottom). Generally speaking, the implementation of node-based programming was straightforward as data was manipulated through temporary objects and loops.

After finishing the both programing files, I moved onto working on my test driver. I immediately realized that the most efficient way to organize the stacked data was to directly create data within the node and array based stacks. In order to accurately test the stack implementation, I used both integer and double data types. Similar to the previous project, I looked onto previous assignments and samples in order to construct the test driver. In doing so, I was able to structure my main file correctly, and tested my implementations with pre-determined data. I implemented additional confirmation statements and organizers in order to clearly read the output and track the execution on the terminal screen, then went back and deleted the placeholders after the debugging process.

Subsequently after a few tests, I was satisfied with my results. Though the instructor did not provide a sample output, he was very clear to what the functionality was supposed to be for each method. By comparing the driver to the previous projects, I was able to map out and correct any possible mistakes. In doing so, I was able to reliably test my code and modify it accordingly. In addition, my confirmation statements allowed me to make sure that every specification was completed and functional.

All in all, the project was very efficient in teaching students about stack classes (templated) and the differences between array and node data-manipulation, while allowing them to further master class implementation through the use of their own test drivers. In doing so, students were able to reflect and strengthen their utilization of pointers and array indexing. Students also learned a different structure of class skeletons that can be used universally and are able to see the strengths and weaknesses of each structure and orientation. If I were to have more time with the project, I would like to be able to modify the stack implementations to be illustrated in a bottom – top format for practical usage. Overall, the project was a success and was a good example to learn from.