George Webster
CSCI 313
HW2—Question 1: Commented Code

Correctly commented code should be legible, and demonstrate a clear understanding of the program and what, and how, it accomplishes its tasks. Commenting on the code for “stack linked list.txt” and “postfix.txt” require going through, and understanding, the code line-by-line. The first file is a stack based on the linked list data type. The second solves postfix mathematical expressions by reading them from a file, evaluating them character by character, and writing the expression and the result into a file.

The file “stack linked list.txt” is known as linkedStack.h in the compiler. It inherits from the stackADT.h abstract data type header file, giving it a number of functions to define and include. The first thing it does is the define node, the basis for all linked lists. Using a template for generic applications and a struct, the node is an object contain some data call info, and a pointer to another node. Then it lists its functions as function headers, before defining them down below. Additionally, it keeps two things private, a pointer to the stack called stackTop, and a function copyStack, which it uses in its copy constructor.

The default constructor merely creates the stack, and assigns stackTop to nullptr. As it is empty, it contains no nodes yet. Then it defines the isEmpty function, called isEmptyStack, which returns true if the stackTop = nullptr. After that, it defines an isFull function, which is required by the abstract data type, but because this stack is based on a linked list, it can never be full, so the Boolean function isFullStack always returns false. Next is the function initializeStack, which returns any stack to an empty condition. Therefore, it serves as a delete function. First it creates a temporary pointer to a node. Then it sets temp = stackTop, advances stackTop along to

the next node, and deletes temp. This continues until stackTop = nullptr, at which point the stack is empty, and therefore initialized.

Next are the three key functions to a stack: pop, top and push. Push, which takes in some element, creates a newNode pointer, then creates the newNode itself. It then stores the element in the node's info variable, and sets the link of newNode to the stackTop. Finally, it sets stackTop equal to newNode. This means that newNode is the new top of the stack. The top function returns the information stored in the stackTop info variable. First it checks that the node is not empty, because if stackTop = nullptr it cannot return anything. Then it simply returns the stackTop ->info. Finally, the pop function removes the top of the stack. To do this, it creates a temporary pointer, and checks, once again, that the stack is not empty by making sure that stackTop points to something. If the stack is not empty, it makes temp equal to stackTop, then moves stackTop to the next node in the list, and then deletes the temporary node. If the stack is empty, it notifies the user and does nothing.

The final function defined is the private function, copyStack. For the process, it creates three pointers, newNode, current, and last. Next, it checks that the stack structure that is being copied into is empty. If it's not empty, it uses initializeStack to delete it. Then it checks if the stack being copied (otherStack) is empty, because if it is, the program can simply stop. Assuming that the stack to be copied is not empty, it sets current to otherStack's stackTop, and copies the stackTop into the new stackTop. Then it sets last point to the new stack's stackTop, and moves the current pointer to the next node in otherStack. Then it creates a while loop, while current does not equal nullptr (i.e. there is more otherStack to be copied). StackTop stays where it is, so both stacks have the same head. NewNode, the final pointer, is used to copy the node of otherStack marked by current, and then last is used to append the newNode to the end of the new

stack linked list. Finally, the last point is moved to the newNode, to allow it to append the next node onto the end, and current is advanced.

Finally, there are the copy constructor, destructor, and overloading the assignment operator. The copy constructor merely makes a new stack, and then uses copyStack to transfer the contents of one stack to the new stack. The destructor uses initializeStack to delete all the nodes in the stack. Finally, the overloaded assignment operator checks that the expression is not saying the stack is equal to itself, as this would create redundancy. Then, it uses the copyStack function to copy otherStack into the assigned stack.

Moving on to “postfix.txt”, which uses a stack and fstream, and four functions and a main. In main, it create a Boolean flag called expressionOk, a character called ch, and a new stack called stack. Then it opens and infile and an outfile stream and attempts to read the text file. If it fails, it reports the issue and terminates the program. Otherwise, it creates a new output file, and sets the writing of the doubles to show the decimal point and be rounded to two places. Then it reads the first character into ch. Finally, it creates a while-loop, allowing it to continue reading multiple expressions from the same file

Inside the while loop, for each expression, it uses intializeStack to erase the stack, sets expressionOk to true, and writes the first character ch to the output file. Then, it calls evaluateExpression, passing it the inputstream, outputstream, stack, character, and expressionOk flag. EvaluateExpression first creates a temporary double to assign numbers read from the input. Then it creates a while loop while the character is not ‘=’, which would indicate we’ve reached the end of the expression. It does a switch statement on ch, checking if it is a number or an operand. If it is the character ‘#’, this means that the following character is a number, so the program reads this into num, writes it into the output file, and then pushes it onto the stack

(checking if the stack is full first). If ch is not a number, then it is an operator, so it call the function evaluateOpr, passing it the outputstream, the stack, the character, and the expressionOk flag.

The evaluateOpr function first creates two doubles, op1 and op2. The it checks if the stack is empty, which it shouldn't be. If it is, it alerts the user and sets the expressionOk flag to false. If it is not empty, it sets op2 equal to the stack.top and then pops the stack. It checks again that the stack is not empty, and sets op1 equal to stack.top and pops the stack again. Then is does a switch statement on the operand, doing the respective evaluation indicated by the symbol, and pushes the result back onto the stack. For division, it checks that op2 is not equal to zero, as this would cause an error, and would set the expressionOk flag to false. If ch is none of +, -, *, or /, then it is not accepted, and the program warns the user and sets the expressionOk flag to false.

Back in evaluateExpression, it check to see if the expressionOk flag is still true. If it is, it reads the next character in the expression, writes it in the output file, and continues the while loop. If expressionOk is false, then it calls the function discardExp, and passes it the inputstream, the outputstream, and the character. The discardExp function allows the program to print out a bad expression without providing an answer, so it doesn't have to terminate the whole program in the case where there may be multiple expressions in the file. It simply says, while the character is not '=', read the character, write it to the output file, and continue.

Finally, back in the while loop in main, the expression has been evaluated, so now printResult is called, and it is passed the outputstream, the stack, and the expressionOk flag. Inside the function, it creates a double called result. Then it checks if the expressionOk flag is still true. If it is, then it goes to the stack. At this point, the stack should contain exactly one number, which is the result. The function checks that the stack is not empty, and sets result equal

to stack.top, then it pops the stack. The stack should now be empty. If it is empty, the result is printed in the output file. If it not, an error has occurred, and the user is notified and the result is not printed. If the stack was received empty by the function, and error has occurred, and the user is notified and the result is not printed. Finally, it puts an underline into the output file to demarcate between one expression and the next. Finally, the program returns to main where it completes the while loop and closes the fstream objects.

This was a fascinating exercise, to follow some pure logic through its entire expression. The error checking was particularly interesting in the postfix part, serving an indication of exactly how much improvement I have to do in this particular area. Developing a deep understanding of another's code is a truly worthwhile exercise in improving one's logic and execution.