```cpp
/*// ------------------------
Justin Dang
Student ID: 1148267

- Uses linked stack code from previous assignment
- Uses String search code from previous assignment(with modifications)

(String search)How to convert char to int:
https://www.geeksforgeeks.org/converting-strings-numbers-cc/

(String search)Understanding the strtok method:
https://stackoverflow.com/questions/5029840/convert-char-to-int-in-c-and-c

(String search)Storing tokens:
https://stackoverflow.com/questions/23970329/how-to-store-tokensstrtok-in-a-pointer-on-an-array

(String search)Converting char[] to double:
https://stackoverflow.com/questions/10605653/converting-char-to-float-or-double

(String search)Understanding isdigit():
http://www.cplusplus.com/reference/cctype/isdigit/


*/// ------------------------

#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cmath>
#include <string.h>
// defines the amount of operators and operands that can be held in total
#define MAX_WORDS 50
// defines the maximum size for userinput and token sizes for each word
#define MAX_SENTENCE_SIZE 256
using namespace std;

// NODE START---------------------------------
/*
NODE blueprint for this project only
*/
class Node {
public:
    double numData;                 // stores double
    char charData;                  // stores char
    Node* next;                     // address of next node(or null/0 to define as end of Queue)
    Node(double info, char info1 = '.', Node* ptr = 0) {
        numData = info;
        charData = info1;
        next = ptr;
    }
}; // NODE END---------------------------------
```

```
51
52  // LINKED QUEUE START--------------------------
53  class Queue {
54  public:
55      Queue() { front = back = 0; }           // constructor that is used to ensure
           our first node and last are set properly
56
57      bool isEmpty() { return front == 0; }   // checks if our first node has a
          address(implying there is another node)
58
59      // info = double, info1 = char
60      void enqueue(double info, char info1 = '.') {
61          Node* temp = new Node(info, info1); // new node stored in temp(note no
              address given to show it is last node)
62
63          if (back == 0)
64              front = back = temp;            // if our last node is null/0 then we
                   create a node that is the front and back
65          else
66          {
67              back->next = temp;              // otherwise we set the address of
                  our last node to our new node
68              back = temp;                    // set our last node = new node
69          }
70      }
71
72      char dequeue() {
73          if (!isEmpty()) {
74              char returnChar = front->charData;
75              Node* temp;                          // temp node where our front node
                       goes
76              temp = front;
77              front = front->next;                 // our element after the first
                  element is now the front element
78              return returnChar;
79          }
80      }
81  private:
82      Node* front, * back;                    // keeps track of front and back of
          queue for traversal
83  }; // LINKED QUEUE END-------------------------------
84
85  // LINKED BASED STACK START--------------------------
86
87  class Stack {
88  private:
89      Node* topOfStack;     // since this is a stack, this will be the only var we
          can interact with
90
91  public:
92      Stack() { topOfStack = 0; }                // creates empty stack
93
```

```
94      bool isEmpty() { return topOfStack == 0; } // returns true if stack is empty
95
96      // info = double, info1 = char
97      void push(double info, char info1 = '.') {
98          Node* temp = new Node(info, info1);    // new node created
99
100         if (isEmpty())
101             topOfStack = temp;                 // if the stack is empty we set
                  the topOfStack directly to new node
102
103         else {
104             temp->next = topOfStack;           // otherwise we grab the address
                  of the new node and set to our top Node
105             topOfStack = temp;                 // we then set the top to the new
                  node
106         }
107     }
108
109     double pop() {
110         if (!isEmpty()) {
111             Node* temp;                            // temp Node used to hold top
                  var and later deleted
112             double returnDouble = topOfStack->numData;
113             temp = topOfStack;                     // set temp to top of stack to
                   delete data of top node
114             topOfStack = topOfStack->next;         // set top Node equal to the
                  next address stored
115             delete temp;                           // delete temp
116             return returnDouble;
117         }
118     }
119 }; // LINKED BASED STACK END--------------------------
120
121 // REVERSE POLISH NOTATION CALCULATOR START------------
122 /*
123 Evaluates a string and if in RPN format, calculates the given equation
124 */
125 class RPN {
126 public:
127     // calculates top of stack or returns -infinity if not supported
128     double calculate(double a, double b, char _operator) {
129         switch (_operator) {   // handles addition, subtraction, division, and
            multiplication,
130         case '+':              // otherwise returns (-infinity)
131             return b + a;
132         case '-':
133             return b - a;
134         case '/':
135             if (a != (double)0)
136                 return b / a;
137             cout << "ERROR: cannot divide by 0\n";
138             return INT_MIN;
```

```
139
140         case '*':
141             return b * a;
142         default:
143             return INT_MIN;     // returns (-infinity)
144         }
145     }
146
147     // orders a char[] into operands and operators if in RPN
148     void sortEquation(char sentence[]) {
149         int num1, num2, i = 0;              // numbers we interact with in stack
150         char* tokens[MAX_WORDS];            // sentence seperated into tokens
              stored here
151
152         // Find tokens in a given char array and store in array
153         for (char* p = strtok(sentence, " "); p; p = strtok(NULL, " ")) {
154             if (i >= 50)
155                 break;
156             tokens[i++] = p;
157         }
158
159         // check if token is num and insert char or operand into stack
160         for (int x = 0; x < i; x++) {
161             char tokenChar = *tokens[x];                    // token is a single
                  char
162             char tokenStr[MAX_SENTENCE_SIZE];               // temp to store
                  token as whole array
163             strcpy(tokenStr, tokens[x]);                    // stores token in
                  char array for accessing
164
165             if (isdigit(tokenStr[strlen(tokenStr) - 1])) {  // checks if char is
                  int
166                 double temp1 = strtod(tokens[x], NULL);     // converts char[] to
                      double
167                 numStack->push(temp1);                      // pushes onto number
                      stack
168                 stackSize++;                                // reflect stack size
                      increase
169             }
170
171             else {
172                 opQueue->enqueue(0, tokenChar);             // pushes into
                    operator stack
173                 queueSize++;                                // reflect queue size
                      increase
174             }
175
176         }
177     }
178     // determines if too many oeprators or too many operands, if not, finds
          results of RPN equation
179     double evaluate(char sentence[]) {
```

```
180         double num1, num2, result;                          // values we interact
               with on stack
181         char op;                                            // value we interact
              with on queue
182         sortEquation(sentence);
183         if ((stackSize - queueSize < 1 && stackSize != 1 + queueSize) ||
              stackSize == 0) {
184             cout << "ERROR: Too many operators\n";           // throws error if
                  too many operators(ends method)
185             return INT_MIN;
186         }
187         else if ((queueSize - stackSize < 1 && stackSize != 1 + queueSize) ||
              queueSize == 0) {
188             cout << "ERROR: Too many operands\n";            // throws error if
                  too many operands(ends method)
189             return INT_MIN;
190         }
191         // if there are the right amount of operands and operators
192         for (int x = 0; x < queueSize; x++) {
193             num1 = numStack->pop();                         // store num1 from
                  stack
194             num2 = numStack->pop();                         // store num2 from
                  stack
195             op = opQueue->dequeue();                        // store op from
                  queue
196             result = calculate(num1, num2, op);             // store result of
                  calc into result
197             numStack->push(result);                         // put the result
                  back into the stack
198         }
199         return result;                                      // return result when
               all calculations are done
200     }
201
202 private:
203     int stackSize = 0, queueSize = 0;
204     Stack* numStack = new Stack();
205     Queue* opQueue = new Queue();
206 };
207
208 // note that output should have "= output"
209 int main()
210 {
211     char sentence[MAX_SENTENCE_SIZE];               // userinput stored here
212     double result;                                  // for ease of access to
          result and code
213
214     cout << "Enter reverse polish equation: ";
215     cin.getline(sentence, MAX_SENTENCE_SIZE);       // user input stored into
          sentence
216     while (sentence[0] != '0') {
217         RPN rpnCalc;                                // creates a new RPN
```

```
            calculator each loop to delete data from previous loop
218         result = rpnCalc.evaluate(sentence);        // store result
219
220         if (result != INT_MIN)                      // determine if result is an
              error or not,
221           cout << "Result: " << result << "\n";    // leading to if the result
                is printed or not
222
223         memset(&sentence[0], 0, sizeof(sentence));  // clearing user input to
              prepare for next loop
224         cout << "\nEnter reverse polish equation: ";
225         cin.getline(sentence, MAX_SENTENCE_SIZE);   // user input stored into
              sentence
226     }
227 }
228 /*//-------------------------------------------case 1:
229 Enter reverse polish equation: 10 15 +
230 Result: 25
231
232 Enter reverse polish equation: 10 15 -
233 Result: -5
234
235 Enter reverse polish equation: 2.5 3.5 +
236 Result: 6
237
238 Enter reverse polish equation: 10 0 /
239 ERROR: cannot divide by 0
240
241 Enter reverse polish equation: 10 20 * /
242 ERROR: Too many operators
243
244 Enter reverse polish equation: 12 20 30 /
245 ERROR: Too many operands
246
247 Enter reverse polish equation: -10 -30 -
248 Result: 20
249
250 Enter reverse polish equation: 100 10 50 25 / * - -2 /
251 Result: 0.15748
252
253 Enter reverse polish equation: 0
254 *///-------------------------------------------
255
```