Assignment One

Josh Seligman

joshua.seligman1@marist.edu

September 11, 2022

1 Singly Linked List

1.1 The Data Structure

A singly linked list is comprised of nodes which contain some form of data as well as a pointer to the next element within the list. As shown in Figure 1.1, the final node has a next of **null**, which marks the end of the list.

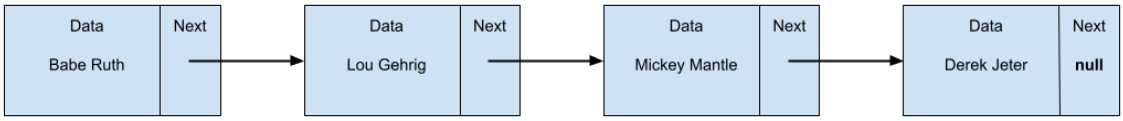

Figure 1.1: Example singly linked list of 4 Yankees legends.

1.2 Asymptotic Analysis

Assuming one only has a pointer to the first element in the list, doing any sort of work with the data requires one to start at the beginning and traverse through the list by using the next pointer within each node until the desired operation is complete. For example, as done in Listing 7 within Section 6.4 on lines 22-26, the only way to dynamically obtain the data for each node was to start at the beginning and iterate through each node by taking advantage of the links. Overall, since linked list operations, such as searching, adding, and removing, all function at a rate of a magnitude of the size of the list, their runtime is classified as $O(n)$.

1.3 Good Parts of the Implementation of a Singly Linked List

1.3.1 Limited Size Restrictions

As previously mentioned in Section 1.1, the last node within a linked list has a next of **null**. This characteristic enables linked lists to have no size restrictions barring memory capacity. As a result, this feature makes linked lists preferred over arrays, which have a fixed length, when the size of the data is frequently changing and

has an unkown maximum. For instance, as demonstrated in Listing 7 of Section 6.4 on lines 13-19, the size of the linked list is only limited by our needs and, if needed, more nodes are able to easily be added to the list with their creation as done on lines 13-15 and linking as shown on lines 18 and 19. On the other hand, if the list was made with an array, the size of the array would have to be provided at the time of creation, and it would not be easy to change the size if additional data have to be added to the array.

1.3.2 Data Type Flexibility

Linked lists do not have to be restricted to be able to store a specific data type. Instead, with the use of generics (C++ templates), the definition of a node is independent of the data type that the user wants to store within the linked list. This provides flexibilty and reusability for many use cases. As demonstrated in Section 6.1 Listing 2, the definition of a node uses a generic T as the type of data being stored, which prevents any assumptions of the data and ensures compatibility with all data types. However, due to how the C++ linker works and to prevent all the code from being written within a single header file, the allowed types have to be stated on lines 13 and 14 in Listing 1. This is a C++ specific issue and is not present in other languages such as Java. Regardless, although they have to be specified for C++, any data type can still be stored within a node and a linked list. A demonstration of the user defining which data type is stored in a node is in Section 6.4 on lines 13-15 of Listing 7. Instead of the Node class defining the data type, the user is able to specify the type of data they want to store, which is a string in this situation but can be anything they want.

2 Stack

2.1 The Data Structure

A stack uses a last in, first out (LIFO) approach to storing data. The most common analogy for stacks is a stack of plates. Each plate is placed on top of each other to build the stack when being stored, but the plate on top is always the first to be used. In other words, the most recent plate that was put away is also the first plate that is taken out. As displayed in Figure 2.1, the stack has a variable called top, which points to the first item in the stack. Additionally, as shown by the arrows between each element, linked lists are used in the implementation of stacks. Stacks have 3 primary functions: push, pop, and isEmpty. Push adds a new element to the top of the stack, pop removes the top item from the stack and returns the item, and isEmpty returns whether or not the stack is empty.

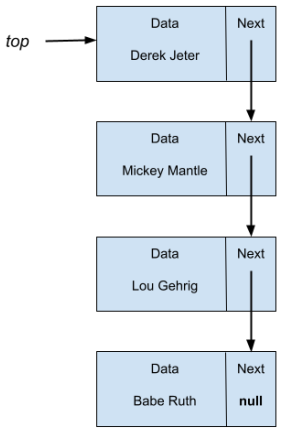

Figure 2.1: Example stack of 4 Yankees legends.

2.2 Asymptotic Analysis

Stacks are incredibly fast when it comes to the implementation of its methods. As demonstrated in Listing 3 in Section 6.2, the push method on lines 14-21 points the new element's next to the current top of the stack (line 17) and updates the top of the stack to point to the new node (line 18). These 2 steps will always run and do not depend on the current size of the stack, which means that the push method runs in $O(1)$ time. The pop method on lines 24-40 is very similar in that it also executes the same few steps regardless of the current size of the stack. These steps are check if the stack is empty for safety (lines 26-29), grab the node from the top of the stack (line 31), update the top pointer to point to the second item in the stack (line 32), update the old top by making it not point to another node (line 36), and return the old top node (line 38). There is no need to traverse the list because the push method made the new node the first element of the list, which is equivalent to the top of the stack. Therefore, just like the push method, the pop method also runs in $O(1)$ time. Lastly, the isEmpty method simply returns the result of the boolean expression on line 45, which also makes it run in $O(1)$ time.

3 Queue

3.1 The Data Structure

A queue uses a first in, first out (FIFO) approach to handling data. One analogy to understand how a queue works is a line to buy movie tickets. The first person that is in line for these tickets is the first to be assisted at the ticket counter. On the other hand, the last person to enter the line will also be the last one to buy their ticket. Similar to stacks, queues are implemented using a singly linked list. As displayed in Figure 3.1, queues have a head variable that points to the first node within the linked list, which equates to the front of the line for the movie tickets. Although not needed to implement a queue, a tail pointer to the last element in the queue may also be provided for efficiency, which is described more in detail in Section 3.2. Lastly, queues have 3 primary functions: enqueue, dequeue, and isEmpty. Enqueue adds a new element to the end of the queue, dequeue removes and returns the first element in the queue, and isEmpty checks to see if the queue is empty or not.

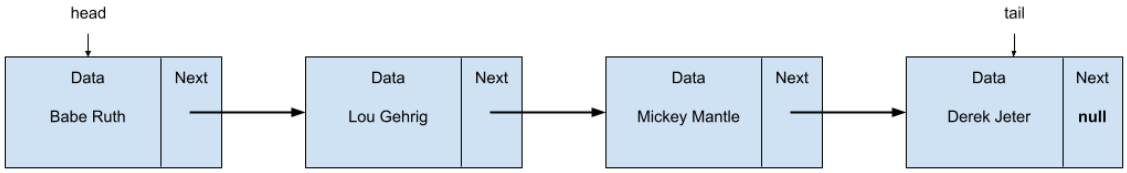

Figure 3.1: Example queue of 4 Yankees legends.

3.2 Asymptotic Analysis

Similar to stacks, queues are also very efficient in the implementation of their respective methods. First, the enqueue method has different run times dependent on the implementation of the data structure. If there is no tail pointer, then one would have to start at the head and then traverse through the list and then set the new node to be the next of the last node in the queue. This implementation results in $O(n)$ runtime because it only loops through the list once. The other implementation of a queue with a tail pointer, despite its need for a little extra memory, greatly reduces the runtime of the enqueue method. As shown in Listing 5 of Section 6.3 on lines 23 and 24, the addition of the tail pointer prevents the need to traverse through the list and, instead, immediately adds the new node to the queue. Since there are now a fixed number of steps regardless of the size of the queue, the new runtime of the enqueue method is $O(1)$. Next, in addition to

the update to the tail pointer if the queue is empty after dequeuing an element on lines 39-42 of Listing 5 of Section 6.3, the dequeue method is identical to the pop method from the Stack class (see Section 2.2), which classifies it as constant time $O(1)$. Lastly, the isEmpty method also runs in $O(1)$ time because it only executes and returns the result of the boolean expression.

4 Good Parts of the Implementation of Stacks and Queues

4.1 Compatibility with the Node Class

As mentioned in Sections 2.1 and 3.1, stacks and queues are both implemented using singly linked lists. Therefore, they both have to be able to utilize the Node class for singly linked lists. On line 9 of Listing 4 in Section 6.2 as well as lines 9 and 12 of Listing 6 in Section 6.3, each of the variables are declared using the template, which defines the data type that the nodes are allowed to use within the stack and queue. The definition of the data type to be used by the stack or queue can be found on lines 32 and 62 of Listing 7 of Section 6.4 at the time of creation of the data structures. This data type is a strict definition for what is being used by the data structure, which means that only that data type can be used. Lines 88 and 89 of Listing 7 in Section 6.4 does not compile because a node with string data cannot be used within a queue for characters, which is why the code is commented out.

4.2 Error Safety

In addition the the type safety of the templates as described in Section 4.1, the Stack and Queue classes also have an error measure to prevent users from breaking the program by doing something that is not allowed. The error safety occurs if the user tries to pop from an empty stack or dequeue from an empty queue. As displayed on lines 26-29 of Listing 3 in Section 6.2 as well as lines 32-35 of Listing 5 in Section 6.3, both the pop and dequeue functions check if the respective data structure is empty and throws an error if it is. This is really important as, without the check, there would be a runtime error that crashes the program from attempting to work with a null pointer. This check impacts how the data structures are used. For instance, in the test program for the queue in Listing 7 of Section 6.4, lines 80-85 use a try-catch block to make sure the programmed error is thrown when attempting to dequeue from the empty queue.

5 Main Program

5.1 Program Overview

The objective of the main program was to print out all the palindromes, ignoring whitespace and capitalization, within a file. More specifically, after normalizing a string for whitespace and capitalization, each character of the string should be both pushed onto a stack and enqueued on a queue. The stack will hold the characters in reverse order and the queue will hold each character in the order as they are in the string. Thus, checking to see if the string is a palindrome requires one to just pop the first character from the stack and dequeue the first character in the queue. If there is ever a time where the characters are different, then the string is not a palindrome. On the other hand, if every comparison is successful, then the string is a palindrome.

5.2 Asymptotic Analysis

The main program contains 2 main components: the file reading and the isPalindrome function. First, the file reading function, as shown in Listing 8 in Section 6.4, consists of 2 loops that each read through the entirety of the file. The first loop (lines 20-24) is used to determine the number of items in the file so the

array can be properly made, and the second loop (lines 39-43) saves the contents of the file into the array. Since each loop iterates for the number of elements in the data and has a fixed number of steps in each iteration, the runtime of the file reading is $O(2n)$, which simplifies to $O(n)$. Next, the isPalindrome function on lines 93-138 of Listing 7 in Section 6.4 has a very similar situation for the runtime complexity. The function begins by iterating through each character of the string and pushing and enqueuing the character to the stack and queue (lines 99-115). Since these functions both run in $O(1)$ time, we can say there are a fixed number of steps for each iteration of the loop. The other loop (lines 117-134) pops and dequeues the stack and queue and performs the comparison described in Section 5.1. Similar to the first loop, since pop and dequeue have runtimes of $O(1)$, there are a fixed number of steps within each iteration of the second loop as well. Therefore, the runtime complexity of isPalindrome, just like the file reading, has an overall runtime complexity of $O(2n)$, which simplifies to $O(n)$.

5.3 Good Parts of the Program's Implementation

5.3.1 The StringArr Struct

As the data has to be read into an array, the program needs access to it outside of the file reading function. Thus, the array needs to be declared on the heap, as shown on line 28 of Listing 8 of Section 6.4, so the array will not be destroyed at the end of the function and the pointer to the array can be used outside the scope of the function. The number of elements of an array in C++ is typically the size of the overall array divided by the size of an individual element. However, since the array was declared on the heap, the length of the array is unkown because we only have the pointer to the first element, which is always 8 bytes. Therefore, as shown on lines 5-9 of Listing 9 in Section 6.4, the definition of the StringArr struct helps with this issue as it stores the pointer to the array as well as the number of elements in the array. This is crucial for the program as it can now loop through each element of the array, which is demonstrated on line 168 of Listing 7 in Section 6.4.

6 Appendix

6.1 Singly Linked List

```
1  #include <string>
2
3  #include "node.h"
4
5  template <typename T>
6  Node<T>::Node(T initialData) {
7      // Initialize the node with the data and without a next node in the linked list
8      Node::data = initialData;
9      Node::next = nullptr;
10 }
11
12 // Define acceptable data types that the Node can accept for the template
13 template class Node<std::string>;
14 template class Node<char>;
```

Listing 1: node.cpp

```
1  #pragma once
2
3  // Node represents an item within a singly linked list and can store data of a given type
4  template <typename T>
5  class Node {
6      public:
7          // A node has the data it is storing (of a type defined by the user) and a pointer
               to the next node
```

```
8           T data;
9
10          // The pointer uses the template to make sure all elements of the linked list store
                the same data type
11          Node<T>* next;
12
13          // Nodes will be instantiated with some data and not have a next node
14          Node(T initialData);
15  };
16
17  // Super helpful resource on templates for c++
18  // https://isocpp.org/wiki/faq/templates#separate-template-fn-defn-from-decl
```

Listing 2: node.h

6.2 Stack

```
1  #include <string>
2  #include <iostream>
3
4  #include "stack.h"
5  #include "node.h"
6
7  // Instantiate the stack with the top pointing to nothing
8  template <typename T>
9  Stack<T>::Stack() {
10     top = nullptr;
11 }
12
13 // Creates a new node and adds it to the stack
14 template <typename T>
15 void Stack<T>::push(Node<T>* newNode) {
16     // Set the next first so we do not lose the rest of the stack
17     newNode->next = top;
18     top = newNode;
19
20     // printStack();
21 }
22
23 // Removes the top node from the stack
24 template <typename T>
25 Node<T>* Stack<T>::pop() {
26     if (isEmpty()) {
27         // Throw an exception if the stack is already empty
28         throw std::invalid_argument("Stack underflow exception. Tried to pop from an empty
               stack.");
29     } else {
30         // We need to collect the data in the node before removing it from the stack
31         Node<T>* topNode = top;
32         top = top->next;
33
34         // We have to remove whatever next is pointing to because the node is no longer
35         // a part of the linked list for the stack
36         topNode->next = nullptr;
37
38         return topNode;
39     }
40 }
41
42 // Checks to see if the stack is empty or not
43 template <typename T>
44 bool Stack<T>::isEmpty() {
45     return top == nullptr;
46 }
47
```

```cpp
template <typename T>
void Stack<T>::printStack() {
    // Get the top of the stack and iterate through, printing the data in each node
    Node<T>* cur = top;
    while (cur != nullptr) {
        std::cout << cur->data << " -> ";
        cur = cur->next;
    }
    // Finish the queue printing
    std::cout << "nullptr" << std::endl;
}

// Define acceptable data types that the Stack can accept for the template
template class Stack<std::string>;
template class Stack<char>;
```

Listing 3: stack.cpp

```cpp
#pragma once

#include "node.h"

template <typename T>
class Stack {
private:
    // Top points to the top of the stack
    Node<T>* top;
public:
    // We need a constructor
    Stack();

    // Push adds a new element to the stack
    void push(Node<T>* newNode);

    // Pop removes the top element from the stack
    Node<T>* pop();

    // isEmpty checks to see if the stack is empty
    bool isEmpty();

    // Prints the stack out
    void printStack();
};
```

Listing 4: stack.h

6.3 Queue

```cpp
#include <string>
#include <iostream>

#include "queue.h"
#include "node.h"

// Instantiate the queue with the head pointing to nothing
template <typename T>
Queue<T>::Queue() {
    head = nullptr;
    tail = nullptr;
}

// Creates a new node and adds it to the queue
template <typename T>
void Queue<T>::enqueue(Node<T>* newNode) {
    if (isEmpty()) {
```

```
18          // Immediately set the head and tail to be the new node if we are empty
19          head = newNode;
20          tail = newNode;
21      } else {
22          // Have the old tail to point to the new node and then update the pointer
23          tail->next = newNode;
24          tail = newNode;
25      }
26      // printQueue();
27  }
28
29  // Removes the front node from the queue
30  template <typename T>
31  Node<T>* Queue<T>::dequeue() {
32      if (isEmpty()) {
33          // Throw an exception if the queue is already empty
34          throw std::invalid_argument("Tried to dequeue from an empty queue.");
35      } else {
36          // We need to collect the data in the node before removing it from the queue
37          Node<T>* frontNode = head;
38          head = head->next;
39          if (head == nullptr) {
40              // Tail has to become nullptr because the queue is now empty
41              tail = nullptr;
42          }
43
44          // We have to remove whatever next is pointing to because the node is no longer
45          // a part of the linked list for the stack
46          frontNode->next = nullptr;
47
48          return frontNode;
49      }
50  }
51
52  // Checks to see if the queue is empty or not
53  template <typename T>
54  bool Queue<T>::isEmpty() {
55      return head == nullptr;
56  }
57
58  template <typename T>
59  void Queue<T>::printQueue() {
60      // Get the head of the queue and iterate through, printing the data in each node
61      Node<T>* cur = head;
62      while (cur != nullptr) {
63          std::cout << cur->data << " -> ";
64          cur = cur->next;
65      }
66      // Finish the queue printing
67      std::cout << "nullptr" << std::endl;
68  }
69
70  // Define acceptable data types that the Queue can accept for the template
71  template class Queue<std::string>;
72  template class Queue<char>;
```

Listing 5: queue.cpp

```
1  #pragma once
2
3  #include "node.h"
4
5  template <typename T>
6  class Queue {
7  private:
```

```
8       // Head points to the front of the queue
9       Node<T>* head;
10
11      // Tail points to the end of the queue (tradeoff to get O(1) enqueue)
12      Node<T>* tail;
13  public:
14      // We need a constructor and destructor
15      Queue();
16
17      // Enqueue adds a new element to the queue
18      void enqueue(Node<T>* newNode);
19
20      // Dequeue removes the front element from the queue
21      Node<T>* dequeue();
22
23      // isEmpty checks to see if the queue is empty
24      bool isEmpty();
25
26      // Prints the entire queue out
27      void printQueue();
28  };
```

Listing 6: queue.h

6.4 Main Program

```
1  #include <iostream>
2  #include <string>
3
4  #include "node.h"
5  #include "stack.h"
6  #include "queue.h"
7  #include "fileUtil.h"
8  #include "util.h"
9
10 // Function to test the Node class
11 void testNode() {
12      // Create the nodes on the stack, so we do not have to delete later
13      Node<std::string> n1("node 1");
14      Node<std::string> n2("node 2");
15      Node<std::string> n3("node 3");
16
17      // Set up the links
18      n1.next = &n2;
19      n2.next = &n3;
20
21      // Print out the data of each node in the linked list
22      Node<std::string>* cur = &n1;
23      while (cur != nullptr) {
24          std::cout << cur->data << std::endl;
25          cur = cur->next;
26      }
27 }
28
29 // Function to test the Stack class
30 void testStack() {
31      // Create a stack and the nodes
32      Stack<char> stack;
33      Node<char> hNode('h');
34      Node<char> sNode('s');
35      Node<char> oNode('o');
36      Node<char> jNode('J');
37
38      // Push each node onto the stack
39      stack.push(&hNode);
```

```
40      stack.push(&sNode);
41      stack.push(&oNode);
42      stack.push(&jNode);
43
44      // Print out the letters as we remove them from the stack
45      while (!stack.isEmpty()) {
46          Node<char>* poppedElem = stack.pop();
47          std::cout << poppedElem->data << '\t' << poppedElem->next << std::endl;
48      }
49      std::cout << std::endl;
50
51      try {
52          // This should throw an error
53          stack.pop();
54      } catch (const std::exception& e) {
55          std::cerr << e.what() << std::endl;
56      }
57  }
58
59  // Function to test the Queue class
60  void testQueue() {
61      // Create a queue and add some data to it
62      Queue<char> queue;
63      Node<char> jNode('J');
64      Node<char> oNode('o');
65      Node<char> sNode('s');
66      Node<char> hNode('h');
67
68      queue.enqueue(&jNode);
69      queue.enqueue(&oNode);
70      queue.enqueue(&sNode);
71      queue.enqueue(&hNode);
72
73      // Print out the letters as we remove them from the queue
74      while (!queue.isEmpty()) {
75          Node<char>* dequeuedElem = queue.dequeue();
76          std::cout << dequeuedElem->data << '\t' << dequeuedElem->next << std::endl;
77      }
78      std::cout << std::endl;
79
80      try {
81          // This should throw an error
82          queue.dequeue();
83      } catch (const std::invalid_argument& e) {
84          std::cerr << e.what() << std::endl;
85      }
86
87      // Proves type safety with templates and fails to compile
88      // Node<std::string> mismatchTest("Hello");
89      // queue.enqueue(mismatchTest);
90  }
91
92  // Function to check if a string is a palindrome, minus whitespace and capitalization
93  bool isPalindrome(std::string word) {
94      // Initialize an empty stack and queue for the checks
95      Stack<char> wordStack;
96      Queue<char> wordQueue;
97
98      // Iterate through each character in the word to populate the stack and queue
99      for (int i = 0; i < word.length(); i++) {
100         char character = word[i];
101         if (character == '␣') {
102             // Go to next character because we are ignoring whitespace
103             continue;
104         } else if (character >= 'a' && character <= 'z') {
```

```
105             // Adjust the character to make it uppercase by taking the difference between
106             // the start of the lowercase letters and the start of the uppercase letters
107             character -= 'a' - 'A';
108         }
109         // Add the character to both the stack and the queue
110         Node<char>* charNodeStack = new Node<char>(character);
111         wordStack.push(charNodeStack);
112 
113         Node<char>* charNodeQueue = new Node<char>(character);
114         wordQueue.enqueue(charNodeQueue);
115     }
116 
117     while (!wordStack.isEmpty() && !wordQueue.isEmpty()) {
118         // Get the character from the top of the stack
119         Node<char>* nodeFromStack = wordStack.pop();
120         char charFromStack = nodeFromStack->data;
121 
122         // Get the character from the top of the queue
123         Node<char>* nodeFromQueue = wordQueue.dequeue();
124         char charFromQueue = nodeFromQueue->data;
125 
126         // We have to delete the nodes from the heap now that we are done with them
127         delete nodeFromStack;
128         delete nodeFromQueue;
129 
130         if (charFromStack != charFromQueue) {
131             // We can return false because we already know that the string is not a
                palindrome
132             return false;
133         }
134     }
135 
136     // The string is a palindrome
137     return true;
138 }
139 
140 int main() {
141     std::cout << "-----␣Testing␣Node␣class␣-----" << std::endl;
142     testNode();
143     std::cout << std::endl;
144 
145     std::cout << "-----␣Testing␣Stack␣class␣-----" << std::endl;
146     testStack();
147     std::cout << std::endl;
148 
149     std::cout << "-----␣Testing␣Queue␣class␣-----" << std::endl;
150     testQueue();
151     std::cout << std::endl;
152 
153     std::cout << "-----␣Testing␣isPalindrome␣-----" << std::endl;
154     std::cout << isPalindrome("racecar") << std::endl; // 1
155     std::cout << isPalindrome("RaCecAr") << std::endl; // 1
156     std::cout << isPalindrome("ra___c␣e___car") << std::endl; // 1
157     std::cout << isPalindrome("4") << std::endl; // 1
158     std::cout << isPalindrome("") << std::endl; // 1
159     std::cout << isPalindrome("ABC") << std::endl; // 0
160     std::cout << std::endl;
161 
162     std::cout << "-----␣Magic␣Items␣-----" << std::endl;
163     try {
164         // Read the file and store it in an array
165         StringArr* data = readFile("magicitems.txt");
166 
167         // Only print out the palindromes
168         for (int i = 0; i < data->length; i++) {
```

```
169             if (isPalindrome(data->arr[i])) {
170                 std::cout << data->arr[i] << std::endl;
171             }
172         }
173 
174         // Clean up memory
175         delete data;
176     } catch (const std::invalid_argument& e) {
177         std::cerr << e.what() << std::endl;
178     }
179 
180     return 0;
181 }
```

Listing 7: main.cpp

```
1  #include <string>
2  #include <fstream>
3  #include <iostream>
4  
5  #include "fileUtil.h"
6  #include "util.h"
7  
8  StringArr* readFile(std::string filePath) {
9      // File reading code from https://cplusplus.com/doc/tutorial/files/
10 
11     // We need to determine how many items there are, so a counter would work well
12     int numItems = 0;
13 
14     // Create the string buffer for storing each line
15     std::string line;
16     // Attempt to open the file
17     std::ifstream file(filePath);
18     if (file.is_open()) {
19         // Read each line from the file
20         while (getline(file, line)) {
21             // During the first read through, we only need to count the number of lines
22             // because we do not know how big to make the array
23             numItems++;
24         }
25 
26         // Create the struct to output the data
27         StringArr* outArr = new StringArr;
28         outArr->arr = new std::string[numItems];
29         outArr->length = numItems;
30 
31         // We will now reread the file and store the data now that the array has been
32         // created with the correct size. Thus, we have to clear any error flags
33         // and fix the pointer to go back to the top of the file
34         file.clear();
35         file.seekg(0);
36 
37         // i is a counter to keep track of the index we are on when reading from the file
38         int i = 0;
39         while (getline(file, line)) {
40             // During the second read through, we will store the contents of the line in the
                   array
41             outArr->arr[i] = line;
42             i++;
43         }
44 
45         // Close the file when done for safety
46         file.close();
47 
48         return outArr;
```

```
    } else {
        // Throw an error if the file was not found
        throw std::invalid_argument("Unable to open file " + filePath);
    }
}
```

Listing 8: fileUtil.cpp

```
#pragma once

// Struct to create string arrays and store their length at the same time
// Especially for use on the heap
struct StringArr {
    std::string* arr;
    int length;
};
```

Listing 9: util.h