

```

1  /*
2   * Programming Challenge 25 - UNIT TEST
3   */
4
5  #include <cassert>
6  #include <iostream>
7  #include <sstream>
8  using namespace std;
9
10 /*
11  * Compute and return the factorial of a value, using a recursive algorithm. Zero factorial
12  * will return 1.
13  * @param value an unsigned integer containing the value for which the factorial will be computed
14  * @return an unsigned integer containing the computed factorial of the value
15  */
16 unsigned int factorial (unsigned int value);
17
18 /*
19  * Return a specified value in a Fibonacci sequence, using a recursive algorithm. The lowest value requested in the sequence
20  * must never be less than one.
21  * @param fibValue an unsigned integer specifying which value in the Fibonacci sequence to return
22  * @return an unsigned integer containing the requested value in the Fibonacci sequence
23  */
24 unsigned int fibonacci (unsigned int fibValue);
25
26 /*
27  * Test a single word to see if it is a palindrome, using a recursive algorithm. The word must be all in the same case
28  * (upper or lower) and cannot contain spaces, digits, or punctuation.
29  * @param word a string containing the word to be tested
30  * @return true if word is a palindrome, else false; empty string and single character strings
31  *         are considered palindromes
32  */
33 bool wordIsPalindrome (string word);
34
35 /*
36  * Produce a string containing the contents of an array, separated by single spaces,
37  * starting at a specified index and going forward to the end of the array, using a recursive algorithm. The returned
38  * string will contain an extra space character after the last value added.
39  * @param array an integer array containing the values to be added to the string
40  * @param startIndex an unsigned integer containing the index of the first value in the array to be added
41  *                  to the output string
42  * @param size an unsigned integer containing the number of elements in the array
43  * @return a string containing the contents of the array, separated by spaces; returns empty string
44  *         if the startIndex is >= the size of the array
45  */
46 string arrayForwardsAsString (int array[], unsigned int startIndex, unsigned int size);
47
48 /*
49  * Produce a string containing the contents of an array, separated by single spaces,
50  * starting at a specified index and going backward to the beginning of the array, using a recursive

```

```

51  * algorithm. The returned string will contain an extra space character after the last value added.
52  * @param array an integer array containing the values to be added to the string
53  * @param startIndex an unsigned integer containing the index of the first value in the array to be added
54  *         to the output string
55  * @param size an unsigned integer containing the number of elements in the array
56  * @return a string containing the contents of the array, separated by spaces, in reverse order; returns empty string
57  *         if the startIndex is < zero
58  */
59  string arrayBackwardsAsString (int array[], int startIndex, unsigned int size);
60
61  /* for unit testing -- do not alter */
62  template <typename X, typename A>
63  void btassert(A assertion);
64  void unittest ();
65
66  int main (int argc, char* argv[])
67  {
68      unittest();
69
70      return 0;
71  }
72
73  /* function declarations */
74
75  unsigned int factorial (unsigned int value)
76  {
77      if( value > 1 )
78      {
79          return (value * factorial ( value - 1 ));
80      }
81      else
82      {
83          return 1;
84      }
85  }
86
87  unsigned int fibonacci (unsigned int fibValue)
88  {
89      if (fibValue == 0 || fibValue == 1)
90      {
91          return fibValue;
92      }
93      else
94      {
95          return fibonacci( (fibValue - 1) ) + fibonacci( (fibValue - 2) );
96      }
97  }
98
99  bool wordIsPalindrome (string word)
100 {

```

```

101     if( word.length() == 0 || word.length() == 1 )
102     {
103         return true;
104     }
105     else if( ( ( word.length() != 0 ) ) && ( word.at( 0 ) == word.at( (word.length() - 1) ) ) )
106     {
107         return wordIsPalindrome( word.substr( 1, (word.length() - 2) ) );
108     }
109     else
110     {
111         return false;
112     }
113 }
114
115 string arrayForwardsAsString (int array[], unsigned int startIndex, unsigned int size)
116 {
117     if( startIndex < size )
118     {
119         stringstream ss;
120         ss << array[startIndex] << " " << arrayForwardsAsString( array, (startIndex + 1), size );
121         return ss.str();
122     }
123
124     return "";
125 }
126
127 string arrayBackwardsAsString (int array[], int startIndex, unsigned int size)
128 {
129     if( startIndex >= 0 )
130     {
131         stringstream ss;
132         ss << array[startIndex] << " " << arrayBackwardsAsString( array, (startIndex - 1), size );
133         return ss.str();
134     }
135
136     return "";
137 }
138
139 /*
140  * Unit testing functions. Do not alter.
141  */
142 void unittest ()
143 {
144     cout << "\nSTARTING UNIT TEST\n\n";
145
146     try {
147         btassert<bool>(factorial(0) == 1);
148         cout << "Passed TEST 1: factorial(0) \n";
149     } catch (bool b) {
150         cout << "# FAILED TEST 1: factorial(0) #\n";

```

```
151     }
152
153     try {
154         btassert<bool>(factorial(1) == 1);
155         cout << "Passed TEST 2: factorial(1) \n";
156     } catch (bool b) {
157         cout << "# FAILED TEST 2: factorial(1) #\n";
158     }
159
160     try {
161         btassert<bool>(factorial(2) == 2);
162         cout << "Passed TEST 3: factorial(2) \n";
163     } catch (bool b) {
164         cout << "# FAILED TEST 3: factorial(2) #\n";
165     }
166
167     try {
168         btassert<bool>(factorial(5) == 120);
169         cout << "Passed TEST 4: factorial(5) \n";
170     } catch (bool b) {
171         cout << "# FAILED TEST 4: factorial(5) #\n";
172     }
173
174     try {
175         btassert<bool>(fibonacci(1) == 1);
176         cout << "Passed TEST 5: fibonacci(1) \n";
177     } catch (bool b) {
178         cout << "# FAILED TEST 5: fibonacci(1) #\n";
179     }
180
181     try {
182         btassert<bool>(fibonacci(2) == 1);
183         cout << "Passed TEST 6: fibonacci(2) \n";
184     } catch (bool b) {
185         cout << "# FAILED TEST 6: fibonacci(2) #\n";
186     }
187
188     try {
189         btassert<bool>(fibonacci(3) == 2);
190         cout << "Passed TEST 7: fibonacci(3) \n";
191     } catch (bool b) {
192         cout << "# FAILED TEST 7: fibonacci(3) #\n";
193     }
194
195     try {
196         btassert<bool>(fibonacci(15) == 610);
197         cout << "Passed TEST 8: fibonacci(15) \n";
198     } catch (bool b) {
199         cout << "# FAILED TEST 8: fibonacci(15) #\n";
200     }
```

```

201
202     try {
203         btassert<bool>(wordIsPalindrome("") == true);
204         cout << "Passed TEST 9: wordIsPalindrome(\"\") \n";
205     } catch (bool b) {
206         cout << "# FAILED TEST 9: wordIsPalindrome(\"\") #\n";
207     }
208
209     try {
210         btassert<bool>(wordIsPalindrome("a") == true);
211         cout << "Passed TEST 10: wordIsPalindrome(\"a\") \n";
212     } catch (bool b) {
213         cout << "# FAILED TEST 10: wordIsPalindrome(\"a\") #\n";
214     }
215
216     try {
217         btassert<bool>(wordIsPalindrome("sitonapotatopanotis") == true);
218         cout << "Passed TEST 11: wordIsPalindrome(\"sitonapotatopanotis\") \n";
219     } catch (bool b) {
220         cout << "# FAILED TEST 11: wordIsPalindrome(\"sitonapotatopanotis\") #\n";
221     }
222
223     try {
224         btassert<bool>(wordIsPalindrome("baseball") == false);
225         cout << "Passed TEST 12: wordIsPalindrome(\"baseball\") \n";
226     } catch (bool b) {
227         cout << "# FAILED TEST 12: wordIsPalindrome(\"baseball\") #\n";
228     }
229
230     int numbers[5] = {5, 10, 15, 20, 25};
231
232     try {
233         btassert<bool>(arrayForwardsAsString(numbers, 0, 5) == "5 10 15 20 25 ");
234         cout << "Passed TEST 13: arrayForwardsAsString(numbers, 0, 5) \n";
235     } catch (bool b) {
236         cout << "# FAILED TEST 13: arrayForwardsAsString(numbers, 0, 5) #\n";
237     }
238
239     try {
240         btassert<bool>(arrayForwardsAsString(numbers, 3, 5) == "20 25 ");
241         cout << "Passed TEST 14: arrayForwardsAsString(numbers, 0, 5) \n";
242     } catch (bool b) {
243         cout << "# FAILED TEST 14: arrayForwardsAsString(numbers, 0, 5) #\n";
244     }
245
246     try {
247         btassert<bool>(arrayForwardsAsString(numbers, 5, 5) == "");
248         cout << "Passed TEST 15: arrayForwardsAsString(numbers, 5, 5) \n";
249     } catch (bool b) {
250         cout << "# FAILED TEST 15: arrayForwardsAsString(numbers, 5, 5) #\n";

```

```
251     }
252
253     try {
254         btassert<bool>(arrayBackwardsAsString(numbers, 4, 5) == "25 20 15 10 5 ");
255         cout << "Passed TEST 16: arrayBackwardsAsString(numbers, 4, 5) \n";
256     } catch (bool b) {
257         cout << "# FAILED TEST 16: arrayBackwardsAsString(numbers, 4, 5) #\n";
258     }
259
260     try {
261         btassert<bool>(arrayBackwardsAsString(numbers, 1, 5) == "10 5 ");
262         cout << "Passed TEST 17: arrayBackwardsAsString(numbers, 1, 5) \n";
263     } catch (bool b) {
264         cout << "# FAILED TEST 17: arrayBackwardsAsString(numbers, 1, 5) #\n";
265     }
266
267     try {
268         btassert<bool>(arrayBackwardsAsString(numbers, -1, 5) == "");
269         cout << "Passed TEST 18: arrayBackwardsAsString(numbers, -1, 5) \n";
270     } catch (bool b) {
271         cout << "# FAILED TEST 18: arrayBackwardsAsString(numbers, -1, 5) #\n";
272     }
273
274     cout << "\nUNIT TEST COMPLETE\n\n";
275 }
276
277 template <typename X, typename A>
278 void btassert (A assertion)
279 {
280     if (!assertion)
281         throw X();
282 }
283
```