

CS 201 Homework 02

Bryan Beus

September 23, 2019

Source Code Link: <https://github.com/siddhartha-crypto/cs201/tree/master/hw2>

1 Design

1.1 Main

For the various improvements, I am curious to implement a hashing function, since I have interest in blockchain technology and encryption.

Beyond this aspect, I think the challenge looks relatively straightforward, based on the provided description.

1.2 Money

This challenge is relatively straightforward. I keep the values in integers until the end, to keep the mathematics simple. After all calculations are complete, I then create the dollar value using a float variable and multiplication.

1.3 Rice

For the Rice challenge, I would like to have the current Square we are calculating visually represented, and the number of rice grains next to this square.

Beyond that simple GUI element, I use only standard print-outs in the console, to keep the task simple.

1.4 Scores

I do not intend to use any design elements for this one, but rather will focus on making sure that I prevent as many bugs as possible.

2 Post Mortem

2.1 Main

One challenge was deciding between the `size_t` data type and the `string` data type for the hash data. I originally wanted to use `string`, since this is the format for the names. However, the boost library that I employed to create the hashes uses `size_t` by default. So, I keep `size_t`, for simplicity's sake.

Another struggle was to match the " " space quotation mark in a call to use `std::string(n, c)` to print a number of spaces after each name before printing the hash. I tried using different methods and types, including `size_t` and `char`, but finally realized that simply using single quotation marks around the space within the `std::string()` function was the best solution.

2.2 Money

This challenge was relatively straightforward and did not take a lot of time. I found that the strings floating around were becoming messy, so I placed them all in vectors to keep things simpler, and only passed the vectors.

I did need to use google to remember how to set the floating point precision numbers. `std::fixed` and `std::setprecision(n)` did the trick.

2.3 Rice

This one appeared deceptively simple. What I thought would be a couple of hours of coding expanded into about six. The complexity grew out of the types of variables, and passing them between the functions.

If I had realized it would become so complicated, I wouldn't have spent time on the visual appearance. This only accounted for maybe an hour total of the time, but still, it was unnecessary.

I did gain a fuller appreciation for the build process with C++. The complexity of declaring variables and types in multiple locations forced me to be accurate in a way that JavaScript would not require. I think that C++ was better at helping me avoid mistakes for this reason.

One more note is that the double value on my computer never had any issues keeping up with the unsigned long long int value. I'm not sure why. Perhaps because my computer is 64 bit?

Also, I had less bugs by declaring the functions first, then writing the code below the `int main()` function. This prevented me from having to check to make sure that a function was declared before it was called.

2.4 Scores

One challenge in the design of this module was to ensure that each vector scaled according to the other.

Sometimes, if I put a name into the names vector, and then fed an improper input (such as a character) to the scores integer, my program design would still record and expand the names vector while preventing the scores vector from expanding.

I added a failsafe that ends the program, in the event that either vector scales without the other matching, and experimented with many potential error paths until I was fairly certain the vectors will scale properly.

3 Answers to Questions

- Typical Sizes
 - CHAR: A char is typically an 8-bit (1 byte) value, ideally used for a single character
 - INT: An int can be from positive to negative 2147483647
 - DOUBLE: A double can handle up to 15 digits, so positive or negative 1.79769 e+308
- #define is a preprocessor macro. It is used to define a block of text and remains defined until the #undef directive is used.
- An initialization is the initial assignment of a value to a variable or data object. An assignment is used at any point in the existence of the variable or data object to change its current value to a potentially different value.
- Technically, when converting a numerical data type from a smaller data type to a type that is capable of holding a wider and larger range of numbers, the conversion can happen easily and automatically. For example, `char a = 1; int b = 0; b = a;` would not struggle in the processor. However, a developer should seek to define conversions as often as possible, to avoid potential issues.
- A computation is any kind of calculation that can include both arithmetical and non-arithmetical steps.
- An expression is a combination of one or more constants, variables, or operators etc. that combine together to compute another value. A statement is a single line of command, and typically ends with a ; semi-colon.
- A constant expression is a value that cannot change. These are used often in programming to ensure that values that should not change, do not change by fault of the programmer.
- On an int you can perform a bitwise operation (<< shift left).

- With a string, you can store alphanumeric characters.
- This initializes a vector of char variables where the initial size of the vector is 20, and is otherwise empty.

4 Sample Output

Listing 1: "Main"

The following hashes belong respectively to the names in the names vector.

```
dorothy:    14174532227680748261
wizard:    15691450355022915147
of:         9028151674594563929
oz:         9549383845444673799
the:        13425634271133782050
man:        6548886666861045578
who:        11765218955868238110
was:        1938579315280203530
thursday:  17603258137849239207
and:        17573395013354419623
```

Press enter to continue...

Listing 2: Money

You have 0 pennies.

You have 1 nickel.

You have 2 dimes.

You have 3 quarters.

The value of all your coins is \$1.00

Press enter to continue...

Listing 3: Rice

```
0
64      9223372036854775808.000000
        9223372036854775808
```

```
Int: Total Grains of Rice Collected:
    2147483647
Int: 1000 Grains of Rice Reached on Square:
    10
Int: 1000000 Grains of Rice Reached on Square:
    20
Int: 1000000000 Grains of Rice Reached on Square:
    30
```

```
Double: Total Grains of Rice Collected:
    1.84467e+19
Double: 1000 Grains of Rice Reached on Square:
    10
Double: 1000000 Grains of Rice Reached on Square:
    20
Double: 1000000000 Grains of Rice Reached on Square:
    30
```

```
Long: Total Grains of Rice Collected:
    18446744073709551615
The above value is the total of all grains of rice on
    all squares.
```

The int value tripped on square: 32

Press enter to continue...

Listing 4: Scores

Please enter names and scores for the database.

To indicate that you are finished entering data, input
a name as "NoName" paired with a score of "0"

```
Enter a new name for the database: Rocky
Enter the score for Rocky: 151
Enter a new name for the database: Dustin
Enter the score for Dustin: 522
Enter a new name for the database: Trevor
Enter the score for Trevor: 322
Enter a new name for the database: Parker
Enter the score for Parker: 151
Enter a new name for the database: NoName
```

Please confirm by entering '0' as a score.

5 My Programs

5.1 Main

```
1 /**
2  * main.cpp
3  * Bryan Beus
4  * CS 201
5  * September 14, 2019
6  * The main program in the assignment - vector names and other
   ↪ features
7  */
8
9 #include <iostream>
10 #include <string>
11 #include <vector>
12 #include <algorithm>
13
```

```

14 // We use the boost library for hashing names
15
16 #include <boost/functional/hash.hpp>
17
18 // Store name inputs from the user
19 // The names variable is declared in the main scope
20
21 void InputNames(std::vector<std::string> & names) {
22
23     // Request 10 names from the user using a for loop
24
25     for (int i = 0; i < 10; i++) {
26
27         std::string name;
28         std::cout << "Please enter a name: ";
29         std::getline (std::cin, name);
30
31         // Place each name in the main scope's names vector
32
33         names.push_back(name);
34     }
35 }
36
37 // Wait for the user to indicate that they are ready to continue
38
39 void waitForContinue() {
40
41     std::cout << std::endl << "Press enter to continue...";
42     getchar();
43 }
44
45 // Clear the console
46
47 void clearConsole() {
48
49     // Clear the console
50
51     std::cout << "\033[2J\033[1;1H";
52 }
53
54
55 // Check whether a user-provided name exists within the names data
56
57 bool DoesNameExist(const std::vector<std::string> & names) {
58
59     // Declare and store the user-provided name
60
61     std::string nameToFind;
62     std::cout << "Tell me a name for which to search in the
        ↪ database: ";
63     std::getline (std::cin, nameToFind);
64
65     // Iterate through the names data to see if the name exists
66     for (int i = 0; i < names.size(); i++) {
67         if (names.at(i) == nameToFind) {

```



```

68
69         // If the name does exist, indicate this in the
        ↪ console and return true
70
71         std::cout << "Yes, the name, " << nameToFind << ", is
        ↪ in the data table." << std::endl;
72         return true;
73     }
74 }
75
76 // If the name does not exist, then indicate this in the
    ↪ console and return false
77
78 std::cout << "No, this name is not in the database." <<
    ↪ std::endl;
79 return false;
80 }
81
82 // Find the length of the longest name of the data
83
84 int getLongestNameLength(const std::vector<std::string> & names)
    ↪ {
85
86     // Test that there is at least one name provided in the names
    ↪ data
87
88     if (names.size() < 1) {
89         std::cout << "Warning: There are no names in the data." <<
            ↪ std::endl;
90         return 0;
91     }
92
93     // Declare the variable and initiate it at the first vector
    ↪ value
94
95     int longest_length = names.at(0).length();
96
97     // Iterate through the names data
98     // If any name is longer than the first name, update the
    ↪ variable to the new longest length
99
100    for (int i = 0; i < names.size(); i++) {
101
102        if (longest_length < names.at(i).length()) {
103            longest_length = names.at(i).length();
104        }
105    }
106
107    // Return the longest length
108
109    return longest_length;
110 }
111

```

```

112 // Iterate through the names data and print each name to the
    ↳ console
113
114 void PrintNames(const std::vector<std::string> & names) {
115
116     // Indicate the stage of the function to the user in the
    ↳ console
117
118     std::cout << "The following names are in the database: " <<
    ↳ std::endl << std::endl;
119
120     // Print each name in the names vector
121
122     for (int i = 0; i < names.size(); i++) {
123
124         std::cout << names.at(i) << std::endl;
125     }
126
127 }
128
129 // Create a hash of each name in the names data, place the hash
    ↳ into a table, and return for later use
130
131 std::vector<std::size_t> CreateHashData(const
    ↳ std::vector<std::string> & names) {
132
133     // Declare the hash table
134
135     std::vector<std::size_t> hash_table;
136
137     // For each name in names, use the boost library to create a
    ↳ hash and place it into the table
138
139     for (int i = 0; i < names.size(); i++) {
140         boost::hash<std::string> string_hash;
141         std::size_t hashed_name = string_hash(names.at(i));
142         hash_table.push_back(hashed_name);
143     }
144
145     // Return the hash_table variable
146
147     return hash_table;
148
149 }
150
151 // Print the names and the associated name hashes in the data
152
153 void PrintNameHashes(const std::vector<std::size_t> & hash_table,
    ↳ const std::vector<std::string> & names) {
154
155     // State in the console what the function does
156
157     std::cout << "The following hashes belong respectively to the
    ↳ names in the names vector." << std::endl << std::endl;

```

```

158
159 // Declare a longest_length variable and use the
    ↳ getLongestNameLength() function to create a universal
    ↳ target length of spaces " "
160
161 int longest_length = getLongestNameLength(names);
162
163 // Iterate through the names and hashes and print them to the
    ↳ table
164 for (int i = 0; i < hash_table.size(); i++) {
165     // Declare the target number of spaces, given the format
    ↳ for the console output
166     // Add 3 units of spaces onto the variable, for good
    ↳ measure
167
168     int num_spaces = longest_length + 3 -
    ↳ names.at(i).length();
169
170     // Print the names, num_spaces spaces between, and the
    ↳ hashes
171
172     std::cout << names.at(i) << ":" << std::string(num_spaces,
173     ↳ ' ') << hash_table.at(i) << std::endl;
174 }
175
176 }
177
178 int main(int argc, char **argv) {
179     // Clear the console
180
181     clearConsole();
182
183     // Declare a names variable to serve throughout the program
184
185     std::vector<std::string> names;
186
187     // Call the InputNames() function to request the user to
    ↳ provide names
188
189     InputNames(names);
190
191     // Wait for user permission to continue
192
193     waitForContinue();
194
195     // Clear the console
196
197     clearConsole();
198
199     // Call the DoesNameExist() function to request the user to
    ↳ search for a name
200
201

```

```

202     DoesNameExist(names);
203
204     // Wait for user permission to continue
205
206     waitForContinue();
207
208     // Clear the console
209
210     clearConsole();
211
212     // Call the PrintNames() function to print names to the
213     ↪ console
214
215     PrintNames(names);
216
217     // Create a vector variable with hashes of the names using
218     ↪ the CreateHashData() function
219
220     std::vector<std::size_t> hash_table = CreateHashData(names);
221
222     // Wait for user permission to continue
223
224     waitForContinue();
225
226     // Clear the console
227
228     clearConsole();
229
230     // Use the PrintNameHashes() function to print all names and
231     ↪ hashes to the console
232
233     PrintNameHashes(hash_table, names);
234
235     // Wait for user permission to continue
236
237     waitForContinue();
238
239     // Clear the console
240
241     clearConsole();
242
243     // End
244
245     return 0;
246 }

```

5.2 Money

```

1  /**
2   * money.cpp
3   * CS 201
4   * Bryan Beus

```

```

5  * September 14, 2019
6  * A program to count the money a user has and to return a clean
   ↳ summation of the value
7  */
8
9  #include <iostream>
10 #include <string>
11 #include <vector>
12 #include <iomanip>
13
14 // Clear the console
15
16 void clearConsole() {
17     // Clear the console
18
19     std::cout << "\033[2J\033[1;1H";
20
21 }
22
23 // Wait for the user to indicate that they are ready to continue
24
25 void waitForContinue() {
26     std::cout << std::endl << "Press enter to continue...";
27     getchar();
28 }
29
30 // Inform the user their input is invalid
31
32 void askUserAgain() {
33     std::cout << "You provided an invalid input. Please try
   ↳ again." << std::endl << std::endl;;
34 }
35
36 // Query the user to input their wallet state
37
38 void queryUserWallet(std::vector<int> & user_wallet,
   ↳ std::vector<std::string> & coin_list_plural) {
39
40     // Declare an input variable for user input
41     int input;
42
43     // Request the user to input the total number of each coin
   ↳ they have in their wallet
44
45     for (int i = 0; i < coin_list_plural.size(); i++) {
46         clearConsole();
47
48         std::cout << "How many " << coin_list_plural.at(i) << "
   ↳ do you have? ";
49
50     }
51
52 }
53
54

```

```

55
56 // Initiate a while loop to wait until the user inputs a
    ↪ viable response
57
58 while (true) {
59
60     std::cin >> input;
61
62     // If the response is invalid, ask again
63
64     if (std::cin.fail() || input < 0) {
65         std::cin.clear();
66         std::cin.ignore(1000, '\n');
67         askUserAgain();
68         waitForContinue();
69
70         // If the response is valid, input the value and
            ↪ move to the next iteration of the for loop
71
72     } else {
73         user_wallet.push_back(input);
74         std::cin.clear();
75         std::cin.ignore(1000, '\n');
76         break;
77     }
78 }
79 }
80 }
81
82 // Calculate the wallet total as a floating point variable
83
84 float calculateWalletTotal(std::vector<int> & user_wallet) {
85
86     float total_wallet = 0;
87
88     // Initiate the various values of the coins
89
90     std::vector<int> values;
91     values.push_back(1);
92     values.push_back(5);
93     values.push_back(10);
94     values.push_back(25);
95
96     // Calculate the total value of the wallet in pennies
97
98     for (int i = 0; i < 4; i++) {
99         total_wallet = total_wallet + user_wallet.at(i) *
            ↪ values.at(i);
100     }
101
102     // Transform the total value into a dollar value
103
104     total_wallet = total_wallet * 0.01;
105

```

```

106     // Return the total value
107
108     return total_wallet;
109 }
110
111 // Print to the console the total wallet sum
112
113 void reportWalletSum(std::vector<int> & user_wallet,
114     ↪ std::vector<std::string> & coin_list_plural,
115     ↪ std::vector<std::string> & coin_list_singular) {
116
117     clearConsole();
118
119     // Call the calculateWalletTotal() function to calculate the
120     ↪ wallet total
121
122     float total_wallet = calculateWalletTotal(user_wallet);
123
124     // For each coin type, print the total in the user's wallet
125     for (int i = 0; i < 4; i++) {
126         std::cout << "You have " << user_wallet.at(i) << " ";
127         if (user_wallet.at(i) == 1) {
128             std::cout << coin_list_singular.at(i);
129         } else {
130             std::cout << coin_list_plural.at(i);
131         }
132         std::cout << "." << std::endl << std::endl;
133     }
134
135     // Print the total value in the wallet
136
137     std::cout << "The value of all your coins is $" << std::fixed
138     ↪ << std::setprecision(2) << total_wallet << std::endl;
139
140     // Pause for user to continue
141
142     waitForContinue();
143 }
144
145 int main(int argc, char **argv) {
146
147     // Clear the console
148
149     clearConsole();
150
151     // Declare the vector to hold the user's coin totals
152
153     std::vector<int> user_wallet;
154
155
156

```

```

157 // Create list of plural coin names
158
159 std::vector<std::string> coin_list_plural;
160 coin_list_plural.push_back("pennies");
161 coin_list_plural.push_back("nickels");
162 coin_list_plural.push_back("dimes");
163 coin_list_plural.push_back("quarters");
164
165 // Create list of singular coin names
166
167 std::vector<std::string> coin_list_singular;
168 coin_list_singular.push_back("penny");
169 coin_list_singular.push_back("nickel");
170 coin_list_singular.push_back("dime");
171 coin_list_singular.push_back("quarter");
172
173 // Query the user's wallet
174
175 queryUserWallet(user_wallet, coin_list_plural);
176
177 // Clear the console
178
179 clearConsole();
180
181 // Report the total value
182
183 reportWalletSum(user_wallet, coin_list_plural,
184 ↪ coin_list_singular);
185
186 return 0;
187 }

```

5.3 Rice

```

1 /**
2  * rice.cpp
3  * CS 201
4  * Bryan Beus
5  * September 15, 2019
6  * A program to display the power of compound interest and to
7  ↪ observe the output in various variable types
8  */
9 #include <iostream>
10 #include <string>
11 #include <vector>
12
13 // Define the Width of the blank spaces in the cells
14 // This must be an even number
15
16 #define Width 6

```



```

17
18 // Clear the console
19
20 void clearConsole();
21
22 // Wait for the user to indicate that they are ready to continue
23
24 void waitForContinue();
25
26 // Set a default function to print a series of blank spaces of
   ↳ length <Width>
27
28 void print_full_width(int longest_length, int col_type);
29
30 // Set a default function to print a series of blank spaces of
   ↳ half of length <Width>
31
32 void print_half_width();
33
34 // Set a default function to print a series of double bars of
   ↳ length <Width>
35
36 void print_full_bar(int longest_length, int col_type);
37
38 // Print the top of the grid
39
40 void print_top_line(int longest_length);
41
42 // Fill a whole row that has no variables or grid corners
43
44 void print_fill_row(int longest_length);
45
46 // Fill a row that has variables, including row numbers and
   ↳ variables inside the grid boxes
47 // Row requires both the current row to print and a vector that
   ↳ has the current state of grid boxes (X's or .'s)
48
49 void printSquare(int & currentSquare);
50
51 // Print a row that will have at least one variable value on it.
52
53 void print_var_row(int & currentSquare, int & longest_length,
   ↳ std::vector<std::string> & current_total_string);
54
55 // Print the bottom line of the grid
56
57 void print_bottom_line(int & longest_length);
58
59 // Print the current square
60
61 void printCurrentSquare(int & currentSquare,
   ↳ std::vector<std::string> & current_total_string);
62
63 // Calculate new values for each important value
64

```

```

65 void calculateNewValues(int & total_in_int, double &
    ↪ total_in_double, unsigned long long int & total_in_long, int
    ↪ & full_total_in_int, double & full_total_in_double, unsigned
    ↪ long long int & full_total_in_long, int & square_int_tripped,
    ↪ int & square_double_tripped);
66
67 // Create a vector of strings that represent the current state of
    ↪ the variables
68 // This helps in formatting the GUI table
69
70 void createTotalString(std::vector<std::string> &
    ↪ current_total_string, int & total_in_int, double &
    ↪ total_in_double, unsigned long long int & total_in_long);
71
72 // Print the measurements for the challenge questions
73
74 void printMeasurements(int & currentSquare, int &
    ↪ full_total_in_int, double & full_total_in_double, unsigned
    ↪ long long int & full_total_in_long, std::vector<int> &
    ↪ values_met_int, std::vector<double> & values_met_double, int
    ↪ & square_int_tripped, int & square_double_tripped, int &
    ↪ total_in_int, double & total_in_double);
75
76 int main(int argc, char **argv) {
77
78     // Declare the variables that represent the grains of rice on
    ↪ a single square for the currently calculated square
79
80     int total_in_int = 1;
81     double total_in_double = 1;
82     unsigned long long int total_in_long = 1;
83
84     // Declare a vector to hold the string representation of the
    ↪ digital values
85     // This is useful for formatting purposes
86
87     std::vector<std::string> current_total_string;
88
89     // Declare variables to represent the sum total of all grains
    ↪ of rice collected
90
91     int full_total_in_int = total_in_int;
92     double full_total_in_double = total_in_double;
93     unsigned long long int full_total_in_long = total_in_long;
94
95     // Declare vectors to track the square numbers at which our
    ↪ challenge questions are met
96
97     std::vector<int> values_met_int;
98     for (int i = 0; i < 3; i++) {
99         values_met_int.push_back(0);
100     }
101

```

```

102     std::vector<double> values_met_double;
103     for (int i = 0; i < 3; i++) {
104         values_met_double.push_back(0);
105     }
106
107     // Declare variables to check when a value type might fail to
108     ↪ keep up with the total numbers
109
110     int square_int_tripped = 0;
111     int square_double_tripped = 0;
112
113     // Declare variables to track the current square and total
114
115     int currentSquare = 1;
116     int totalSquares = 64;
117
118     // Clear the console before we begin
119     clearConsole();
120
121     // Initiate a while loop for all calculations and displays
122
123     while (currentSquare <= totalSquares) {
124
125         // Call the createTotalString function to create the
126         ↪ string representations of our grains of rice on the
127         ↪ current square
128
129         createTotalString(current_total_string, total_in_int,
130         ↪ total_in_double, total_in_long);
131
132         // Display the current square
133
134         printCurrentSquare(currentSquare, current_total_string);
135
136         // Print the measurements that track our challenge
137         ↪ questions
138
139         printMeasurements(currentSquare, full_total_in_int,
140         ↪ full_total_in_double, full_total_in_long,
141         ↪ values_met_int, values_met_double,
142         ↪ square_int_tripped, square_double_tripped,
143         ↪ total_in_int, total_in_double);
144
145         // Wait for the user to indicate they are ready to proceed
146         ↪ to the next square
147
148         waitForContinue();
149
150         // Clear the console before proceeding
151         clearConsole();
152
153         // Calculate the values for the next square

```

```

146         calculateNewValues(total_in_int, total_in_double,
            ↪ total_in_long, full_total_in_int,
            ↪ full_total_in_double, full_total_in_long,
            ↪ square_int_trippled, square_double_trippled);
147
148         // Increase our total square count
149
150         ++currentSquare;
151     }
152
153     return 0;
154 }
155
156 // Clear the console
157
158 void clearConsole() {
159     std::cout << "\033[2J\033[1;1H";
160 }
161
162 // Wait for the user to indicate that they are ready to continue
163
164 void waitForContinue() {
165     std::cout << std::endl << "Press enter to continue...";
166     getchar();
167 }
168
169 // Set a default function to print a series of blank spaces of
170 ↪ length <Width>
171
172 void print_full_width(int longest_length, int col_type) {
173     // If the column is on the left, print a Width-wide row of
174     ↪ blank spaces
175
176     if (col_type == 0) {
177         for (int i = 0; i < Width; i++) {
178             std::cout << " ";
179         }
180     }
181
182     // If the column is on the right, print a row of blank spaces
183     ↪ that appropriately matches the length of the longest
184     ↪ number of grains of rice
185
186     } else if (col_type == 1)
187         for (int i = 0; i < longest_length + (Width * 2 / 3); i++)
188             {
189                 std::cout << " ";
190             }
191     }
192 }

```

```

191 // Set a default function to print a series of blank spaces of
    ↳ half of length <Width>
192
193 void print_half_width() {
194     for (int j = 0; j < (Width * 1 / 3); j++) {
195         std::cout << " ";
196     }
197 }
198
199 // Set a default function to print a series of double bars of
    ↳ length <Width>
200
201 void print_full_bar(int longest_length, int col_type) {
202     // If the column is on the left, print a bar of Width length
203
204     if (col_type == 0) {
205         for (int i = 0; i < Width; i++) {
206             std::cout << " ";
207         }
208     }
209
210     // If the column is on the right, print a bar of a length
    ↳ appropriate for the longest number of grains of rice
211
212     } else if (col_type == 1)
213         for (int i = 0; i < longest_length + (Width * 2 / 3); i++)
    ↳ {
214         std::cout << " ";
215     }
216 }
217
218 // Print the top of the grid
219
220 void print_top_line(int longest_length) {
221     // Vertically clear at least one line in the terminal, then
    ↳ print the <Width> blank spaces
222
223     std::cout << std::endl;
224
225     // Print the top row of the grid
226
227     std::cout << " ";
228
229     print_full_bar(longest_length, 0);
230
231     std::cout << " ";
232
233     print_full_bar(longest_length, 1);
234
235     std::cout << " " << std::endl;
236 }
237
238
239 // Fill a whole row that has no variables or grid corners

```

```

240
241 void print_fill_row(int longest_length) {
242     // Print a divider bar with some formatting spaces
243     std::cout << " ";
244     // Call the print_full_width() function to print the left
245     ↪ column
246     print_full_width(longest_length, 0);
247     // Print a divider bar
248     std::cout << " ";
249     // Call the print_full_width() function to print the right
250     ↪ column
251     print_full_width(longest_length, 1);
252     // Print a divider bar
253     std::cout << " " << std::endl;
254 }
255 // Fill a row that has variables, including row numbers and
256 ↪ variables inside the grid boxes
257 void printSquare(int & currentSquare) {
258     // Call default function to print half width of spaces
259     print_half_width();
260     // If the current square number is less than 10, add an extra
261     ↪ space for formatting
262     if (currentSquare < 10) {
263         std::cout << " ";
264     }
265     // Print the current square number
266     std::cout << currentSquare;
267     // Call default function to print half width of spaces
268     print_half_width();
269 }
270
271

```

```

290 // Print a row in the rice/square GUI element that has variables
    ↳ on it
291
292 void print_var_row(int & currentSquare, int & longest_length,
    ↳ std::vector<std::string> & current_total_string) {
293
294     // Iterate through each of the rows
295
296     for (int i = 0; i < 3; i++) {
297
298         // Print the first divider bar
299
300         std::cout << " ";
301
302         // If this is the second row, print the square number
303
304         if (i == 1) {
305             printSquare(currentSquare);
306
307             // Otherwise, keep the first column blank
308
309         } else {
310             print_full_width(longest_length, 0);
311         }
312
313         // Divider bar
314
315         std::cout << " ";
316
317         // Print a bit of extra space for formatting, before
            ↳ printing rice grain numbers
318
319         print_half_width();
320
321         // Check how many blank spaces are needed to keep the
            ↳ current number in sync with the format of the grid
322
323         int num_spaces = longest_length -
            ↳ current_total_string.at(i).length();
324
325         // Print the number of grains of rice, and the necessary
            ↳ blank spaces for formatting
326         std::cout << current_total_string.at(i) <<
            ↳ std::string(num_spaces, ' ');
327
328         // Print some more padding
329
330         print_half_width();
331
332         // Final divider bar
333
334         std::cout << " " << std::endl;
335     }
336

```

```

337 }
338
339 // Print the bottom line of the grid
340
341 void print_bottom_line(int & longest_length) {
342     // Print bottom corner
343
344     std::cout << " ";
345
346     // Print a full bar of appropriate length for left column
347     print_full_bar(longest_length, 0);
348
349     // Print divider
350
351     std::cout << " ";
352
353     // Print a full bar of appropriate length for right column
354     print_full_bar(longest_length, 1);
355
356     // Print right bottom corner
357
358     std::cout << " " << std::endl;
359 }
360
361 // Print the current square number, with the appropriate number
362 // of empty spaces around it
363
364 void printCurrentSquare(int & currentSquare,
365     ↪ std::vector<std::string> & current_total_string) {
366
367     // Calculate the longest length of the three records
368
369     int longest_length = current_total_string.at(0).length();
370
371     for (int i = 1; i < current_total_string.size(); i++) {
372
373         if (longest_length < current_total_string.at(i).length())
374             ↪ {
375                 longest_length = current_total_string.at(i).length();
376             }
377     }
378
379     // Print first rows of grid
380
381     print_top_line(longest_length);
382     print_fill_row(longest_length);
383
384     // Print the variable rows
385
386     print_var_row(currentSquare, longest_length,
387         ↪ current_total_string);

```



```

388     // the bottom rows of grid
389
390     print_fill_row(longest_length);
391     print_bottom_line(longest_length);
392
393 }
394
395 // Calculate new values for each of the important variables; Call
    ↪ this function after printing the current variables to the
    ↪ console
396
397 void calculateNewValues(int & total_in_int, double &
    ↪ total_in_double, unsigned long long int & total_in_long, int
    ↪ & full_total_in_int, double & full_total_in_double, unsigned
    ↪ long long int & full_total_in_long, int & square_int_trippled,
    ↪ int & square_double_trippled) {
398
399     // Double the current values of grains of rice on the square
400
401     total_in_int = 2 * total_in_int;
402     total_in_double = 2 * total_in_double;
403     total_in_long = 2 * total_in_long;
404
405     // While ensuring that we're not adding negatives or zeros
    ↪ (should the size increase beyond capacity), add the
    ↪ current square's rice to the running total for each
    ↪ variable type
406
407     if (total_in_int >= 1) {
408         full_total_in_int = full_total_in_int + total_in_int;
409     }
410
411     if (total_in_double >= 1) {
412         full_total_in_double = full_total_in_double +
    ↪ total_in_double;
413     }
414
415     if (total_in_long >= 1) {
416         full_total_in_long = full_total_in_long + total_in_long;
417     }
418
419 }
420
421 // Create a string that can visually represent the state of the
    ↪ current square's grains of rice count
422 // This is useful for formatting
423
424 void createTotalString(std::vector<std::string> &
    ↪ current_total_string, int & total_in_int, double &
    ↪ total_in_double, unsigned long long int & total_in_long) {
425
426     // Clear the current_total_string vector
427

```

```

428     current_total_string.clear();
429     // Add in the new numbers as strings
430     current_total_string.push_back(std::to_string(total_in_int));
431     ↪ current_total_string.push_back(std::to_string(total_in_double));
432     current_total_string.push_back(std::to_string(total_in_long));
433 }
434 // Print the current measurements that answer the challenge
435 ↪ questions
436 void printMeasurements(int & currentSquare, int &
437     full_total_in_int, double & full_total_in_double, unsigned
438     ↪ long long int & full_total_in_long, std::vector<int> &
439     ↪ values_met_int, std::vector<double> & values_met_double, int
440     ↪ & square_int_tripped, int & square_double_tripped, int &
441     ↪ total_in_int, double & total_in_double) {
442     // For each data type and for each of the three standards
443     ↪ that we want to measure in the challenge questions, check
444     ↪ to see whether or not we have surpassed that number of
445     ↪ grains of rice
446     // If we have, add this value to our vector that tracks the
447     ↪ square on which this event occurs
448     if (full_total_in_int >= 1000 && values_met_int.at(0) == 0) {
449         values_met_int.at(0) = currentSquare;
450     }
451     if (full_total_in_int >= 1000000 && values_met_int.at(1) ==
452     ↪ 0) {
453         values_met_int.at(1) = currentSquare;
454     }
455     if (full_total_in_int >= 1000000000 && values_met_int.at(2)
456     ↪ == 0) {
457         values_met_int.at(2) = currentSquare;
458     }
459     if (full_total_in_double >= 1000 && values_met_double.at(0)
460     ↪ == 0) {
461         values_met_double.at(0) = currentSquare;
462     }
463     if (full_total_in_double >= 1000000 &&
464     ↪ values_met_double.at(1) == 0) {
465         values_met_double.at(1) = currentSquare;
466     }

```

```

466
467 if (full_total_in_double >= 1000000000 &&
    ↪ values_met_double.at(2) == 0) {
468     values_met_double.at(2) = currentSquare;
469 }
470
471 // Add an extra space for formatting
472
473 std::cout << std::endl;
474
475 // Print the running total of grains of rice, according to
    ↪ the int data type
476
477     std::cout << "Int: Total Grains of Rice Collected:
    ↪ " << full_total_in_int << std::endl;
478
479 // For the int data type, for each of the three standards we
    ↪ measure, when they occur print them to the console
480
481 if (values_met_int.at(0) > 0) {
482     std::cout << "Int: 1000 Grains of Rice Reached on Square:
    ↪ " << values_met_int.at(0) << std::endl;
483 }
484
485 if (values_met_int.at(1) > 0) {
486     std::cout << "Int: 1000000 Grains of Rice Reached on
    ↪ Square: " << values_met_int.at(1) << std::endl;
487 }
488
489 if (values_met_int.at(2) > 0) {
490     std::cout << "Int: 1000000000 Grains of Rice Reached on
    ↪ Square: " << values_met_int.at(2) << std::endl <<
    ↪ std::endl;
491 }
492
493 // Add an extra space for formatting
494
495 std::cout << std::endl;
496
497 // Print the running total of grains of rice, according to
    ↪ the double data type
498
499     std::cout << "Double: Total Grains of Rice Collected:
    ↪ " << full_total_in_double << std::endl;
500
501 // For the double data type, for each of the three standards
    ↪ we measure, when they occur print them to the console
502
503 if (values_met_double.at(0) > 0) {
504     std::cout << "Double: 1000 Grains of Rice Reached on
    ↪ Square: " << values_met_double.at(0) <<
    ↪ std::endl;
505 }

```

```

506
507 if (values_met_double.at(1) > 0) {
508     std::cout << "Double: 1000000 Grains of Rice Reached on
        ↳ Square: " << values_met_double.at(1) << std::endl;
509 }
510
511 if (values_met_double.at(2) > 0) {
512     std::cout << "Double: 1000000000 Grains of Rice Reached
        ↳ on Square: " << values_met_double.at(2) << std::endl
        ↳ << std::endl;
513 }
514
515 // Add an extra space for formatting
516
517 std::cout << std::endl;
518
519 // Print the running total in the long data type
520
521     std::cout << "Long: Total Grains of Rice Collected:
        ↳ " << full_total_in_long << std::endl;
522
523 // When we reach the end of all calculations, print our
        ↳ result in the console
524
525 if (currentSquare == 64) {
526     std::cout << "The above value is the total of all grains
        ↳ of rice on all squares." << std::endl;
527 }
528
529 // Add an extra space for formatting
530
531 std::cout << std::endl;
532
533 // Calculate the square on which the int or the double data
        ↳ type may stop keeping up with our running total
534
535 if (square_int_tripped == 0 && total_in_int <= 0) {
536     square_int_tripped = currentSquare;
537 }
538
539 if (square_double_tripped == 0 && total_in_double <= 0) {
540     square_double_tripped = currentSquare;
541 }
542
543 // Report the square on which any failed data type
        ↳ experienced the failure
544
545 if (square_int_tripped != 0) {
546
547     std::cout << "The int value tripped on square: " <<
        ↳ square_int_tripped << std::endl;
548 }
549

```

```

550     if (square_double_tripped != 0) {
551         std::cout << "The double value tripped on square: " <<
552             ↪ square_int_tripped;
553     }
554 }
555 }

```

5.4 Scores

```

1  /**
2   * scores.cpp
3   * CS 201
4   * Bryan Beus
5   * September 18, 2019
6   * A program to record names and scores in two separate vectors
7   */
8
9  #include <iostream>
10 #include <vector>
11 #include <string>
12 #include <algorithm>
13
14 using std::cout;
15 using std::endl;
16 using std::string;
17 using std::vector;
18 using std::cin;
19
20 // Clear the console
21
22 void clearConsole();
23
24 // Wait for user input
25
26 void waitForContinue();
27
28 // Display the main prompt screen
29
30 bool displayPrompt(vector<string> & names, vector<int> & scores);
31
32 // Request the user to input a name and score
33
34 bool addInput(vector<string> & names, vector<int> & scores);
35
36 // Request a name
37
38 bool requestName(vector<string> & names, string & newName);
39
40 // Request a score
41

```

```

42 bool requestScore(vector<int> & scores, int & newScore, string &
   ↪ newName);
43
44 // Check that the inputted name is not a duplicate
45
46 bool checkOriginal(string & newName, vector<string> & names);
47
48 // Check that the lengths of the vectors match
49
50 bool checkLengths(vector<string> & names, vector<int> & scores);
51
52 // Check that the database is not empty
53
54 bool checkDatabaseHasInputs(vector<string> & names);
55
56 // Print a list of names and scores
57
58 bool printList(vector<string> & names, vector<int> & scores, bool
   ↪ wait);
59
60 // Search for a name in the database
61
62 void searchName(vector<string> & names, vector<int> & scores);
63
64 // Search for a score in the database
65
66 void searchScore(vector<string> & names, vector<int> & scores);
67
68 int main(int argc, char **argv) {
69
70     // Declare names and scores vectors
71
72     vector<string> names;
73     vector<int> scores;
74
75     // Initiate endless while loop to maintain program stream
76
77     while (true) {
78
79         // Initiate bool variable to allow program to end, if
   ↪ needed
80         // Call main displayPrompt() function to begin program
81
82         bool result = displayPrompt(names, scores);
83
84         // If result is ever returned negative, end the program
85
86         if (!result) {
87             break;
88         }
89     }
90
91     return 0;
92 }
93

```

```

94 // Clear the console
95
96 void clearConsole() {
97     cout << "\033[2J\033[1;1H";
98 }
99
100 // Wait for user input
101
102 void waitForContinue() {
103     cout << "Press enter to continue . . . ";
104     getchar();
105
106     // Clear cin
107
108     cin.clear();
109     cin.ignore(1000, '\n');
110 }
111
112 // Display the main prompt screen
113
114 bool displayPrompt(vector<string> & names, vector<int> & scores)
115 ↪ {
116     // Clear the console
117
118     clearConsole();
119
120     // Declare a result variable that will be sent back to the
121     ↪ int main() function's while loop
122
123     bool result = true;
124
125     // Declare option variable to capture user input for program
126     ↪ direction
127
128     int option;
129
130     // Display options
131
132     cout << "Choose an option from the following menu: " << endl;
133     cout << "\n1) Add new names and scores to the database" << endl;
134     cout << "2) Print the full list of names and scores" << endl;
135     cout << "3) Search for a name" << endl;
136     cout << "4) Search for a score" << endl;
137     cout << "0) End program\n" << endl;
138
139     // Initiate endless while loop
140     // Loop continues until user provides valid input
141
142     while (true) {
143         // Capture input
144
145         cin >> option;

```

```

146         // Ensure input is valid
147
148     if (cin.fail() || option > 4 || option < 0) {
149
150         // Clear cin
151
152         cin.clear();
153         cin.ignore(1000, '\n');
154         cout << "\nThe option you selected is not valid.
155             ↪ Please try again: ";
156
157         // If the input is valid, then break the loop to
158         ↪ continue
159     } else {
160         break;
161     }
162
163     // Initiate switch method to determine response to input
164
165     switch (option) {
166
167         // Add an input
168
169     case 1:
170         result = addInput(names, scores);
171         break;
172
173         // Print a list of names
174
175     case 2:
176         printList(names, scores, true);
177         break;
178
179         // Search for a name in the database
180
181     case 3:
182         searchName(names, scores);
183         break;
184
185         // Search for a score in the database
186
187     case 4:
188         searchScore(names, scores);
189         break;
190
191         // End the program by returning false
192
193     case 0:
194         result = false;
195         break;
196
197         // Restart stream by default
198

```



```

199     default:
200         break;
201     }
202     // Return the result to the endless while loop in int main()
203
204     return result;
205 }
206
207 // Request the user to input a name and score
208
209 bool addInput(vector<string> & names, vector<int> & scores) {
210     // Declare result variable to monitor user progress, and if
211     ↪ necessary, return a false value, and thus end the program
212
213     bool result = true;
214
215     // Clear the console
216
217     clearConsole();
218
219     // Print request for user input
220
221     cout << "Please enter names and scores for the database." <<
222     ↪ endl;
223     cout << "\nTo indicate that you are finished entering data,
224     ↪ input a name as \"NoName\" paired with a score of \"0\"\\n"
225     ↪ << endl;
226
227     // Initiate endless while loop to request new name
228
229     while (true) {
230         // Declare newName variable
231         // Declare newScore variable
232
233         string newName;
234         int newScore;
235
236         // Declare result variable to ensure call to
237         ↪ requestName() function is successful
238         // Call requestName() function
239
240         result = requestName(names, newName);
241
242         // If the result is unsuccessful, end the loop (and
243         ↪ program)
244
245         if (!result) {
246             break;
247
248             // Otherwise, call the requestScore() function and set
249             ↪ the response to the result variable

```

```

246     } else if (result) {
247         result = requestScore(scores, newScore, newName);
248     }
249
250     // Check to see if the input values are NoName and 0
251     // If so, end the while loop
252
253     if (newName == "NoName" && newScore == 0) {
254         break;
255     }
256 }
257
258 // Check that the length of the names and scores vectors are
259 // the same
260 // If not, end the program with error message
261
262 if (!checkLengths(names, scores)) {
263     cout << "The names and scores vectors are of different
264         lengths. Something is wrong in the code." << endl;
265     result = false;
266 }
267
268 // Return the result
269 return result;
270 }
271
272 // Request a name for the names vector
273
274 bool requestName(vector<string> & names, string & newName) {
275     // Declare the result variable to return at the end
276
277     bool result = true;
278
279     // Print request for new name
280
281     cout << "Enter a new name for the database: ";
282
283     // Initiate endless while loop to request new name
284
285     while (true) {
286         // Capture new name
287
288         cin >> newName;
289
290         // If the input type is invalid, start again
291
292         if (cin.fail()) {
293             cin.clear();
294             cin.ignore(1000, '\n');
295             cout << "\nThe input you provided is not valid. Please
296                 try again.\n" << endl;

```

```

298
299         // Check that this input name is original
300         // If it is not, end the program
301
302     } else if (!checkOriginal(newName, names)) {
303         cin.clear();
304         cin.ignore(1000, '\n');
305         cout << "\nThe name you provided is already in the
        ↳ database. Terminating program (as per
        ↳ instructions)." << endl;
306         result = false;
307         break;
308
309         // Check to see if the user is beginning the
        ↳ termination process
310
311     } else if (newName == "NoName") {
312         break;
313
314         // Add the name to the database
315     } else {
316         names.push_back(newName);
317         break;
318     }
319 }
320
321 // Return the result
322
323 return result;
324 }
325
326 // Request a new score
327
328 bool requestScore(vector<int> & scores, int & newScore, string &
    ↳ newName) {
329
330     // Declare result variable to monitor function progress
331
332     bool result = true;
333
334     // Check to see if the current newName variable is NoName
335     // If it is, request user to confirm by entering 0
336     if (newName == "NoName") {
337
338
339         cout << "\nPlease confirm by entering '0' as a score.\n"
        ↳ << endl;
340
341         // Initiate endless while loop to capture valid user
        ↳ response
342
343         while (true) {
344
345             // Capture user input

```

```

346         cin >> newScore;
347
348         // Verify user input
349
350         if (cin.fail()) {
351             cin.clear();
352             cin.ignore(1000, '\n');
353             cout << "The input provided is not a valid
354                 ↪ integer. Please try again." << endl;
355
356             // If the user input is 0, return to the main
357             ↪ display prompt
358         } else if (newScore == 0) {
359             return result;
360
361             // If the user enters a valid input other than 0,
362             ↪ continue putting new names and scores into
363             ↪ the database
364         } else {
365             cout << "\nContinuing with name and score database
366                 ↪ inputs.\n" << endl;
367             break;
368         }
369     }
370
371     // If the newName variable is not 'NoName,' begin the process
372     ↪ for collecting a matching score
373 } else {
374     // Request new score
375     cout << "Enter the score for " << newName << ": ";
376
377     // Initiate endless while loop to capture valid user score
378     while (true) {
379         // Capture new score value
380
381         cin >> newScore;
382
383         // If input is invalid restart the loop
384
385         if (cin.fail()) {
386             cin.clear();
387             cin.ignore(1000, '\n');
388             cout << "The input provided is not valid. Please
389                 ↪ try again: ";
390
391
392

```

```

393         // Otherwise, add the new score to the database and
394         ↪ end the loop
395     } else {
396
397         scores.push_back(newScore);
398         break;
399     }
400 }
401 }
402 }
403
404 // Return the result variable
405
406 return result;
407
408 }
409
410 // Check that the input newName variable is not a duplicate
411
412 bool checkOriginal(string & newName, vector<string> & names) {
413
414     // Declare isOriginal variable to check whether newName is new
415
416     bool isOriginal = true;
417
418     // Iterate through names vector to check for duplicates
419
420     for (int i = 0; i < names.size(); i++) {
421
422         // If a duplicate is found, set isOriginal to false
423         if (names.at(i) == newName) {
424             isOriginal = false;
425         }
426     }
427
428     // Return result
429
430     return isOriginal;
431 }
432
433 // Check to make sure that names and scores vectors are valid
434
435 bool checkLengths(vector<string> & names, vector<int> & scores) {
436
437     // Check that the lengths are correct, and return a bool
438     ↪ result
439
440     bool isCorrect = (names.size() == scores.size()) ? true :
441     ↪ false;
442
443     // Return the bool result
444
445     return isCorrect;

```

```

445 }
446
447 // Check that the database of names is not empty
448
449 bool checkDatabaseHasInputs(vector<string> & names) {
450     // If size of names vector is less than one, return
451     ↪ instructions to user, wait for user to confirm, and then
452     ↪ end the function with a negative
453
454     if (names.size() < 1) {
455         cout << "You must put names and scores in the database
456         ↪ before attempting to read it." << endl;
457         waitForContinue();
458         return false;
459     }
460
461     // If the size is greater than or equal to one, return
462     ↪ positive
463     return true;
464 }
465
466 // Print a list of provided names and scores
467 bool printList(vector<string> & names, vector<int> & scores, bool
468 ↪ wait) {
469     // Clear the console
470
471     clearConsole();
472
473     // Check that the database is not empty
474
475     if (checkDatabaseHasInputs(names)) {
476         // Print the names and scores in columns
477
478         while (true) {
479             // Declare names and scores title strings
480
481             string columnOne = "Names";
482             string columnTwo = "Scores";
483
484             // Declare variable for standard char size of column
485             ↪ one
486
487             int columnSize = columnOne.length() + 3;
488
489             // Iterate through list of names and discover longest
490             ↪ column one name
491
492             for (int i = 0; i < names.size(); i++) {

```

```

493         if (names.at(i).length() > columnSize + 3) {
494             columnSize = names.at(i).length() + 3;
495         }
496     }
497
498     // Declare variables to manage column formatting
499
500     int columnTitleSpaces = columnSize -
501     ↪ columnOne.length();
502     int dashCount = columnTwo.length() + columnSize;
503
504     // Print column titles
505
506     cout << columnOne <<
507     ↪ string(std::max(columnTitleSpaces, 3), ' ') <<
508     ↪ columnTwo << endl;
509     cout << string(dashCount, '-') << endl;
510
511     // Print all names and scores
512
513     for (int i = 0; i < names.size(); i++) {
514         cout << names.at(i) << string(columnSize -
515         ↪ names.at(i).length(), ' ') << scores.at(i) <<
516         ↪ endl;
517     }
518
519     // Wait for user to continue
520
521     if (wait) {
522         waitForContinue();
523     }
524
525     // End while loop
526     break;
527 }
528 }
529
530 // Search for a name in the database
531 void searchName(vector<string> & names, vector<int> & scores) {
532     // Clear the console
533     clearConsole();
534
535     // Declare searchName variable
536     // Initiate nameFound variable and begin with default at
537     ↪ negative
538
539     string searchName;
540     bool nameFound = false;

```

```

541
542 // Declare a tempName and tempScore variable
543 // If name(s) and score(s) are found, these variable are sent
    ↳ to the printList() function
544
545 vector<string> tempName;
546 vector<int> tempScore;
547
548 // Verify that the database is not empty
549
550 if (checkDatabaseHasInputs(names)) {
551
552 // Print request for user input
553
554 cout << "Enter the name for which to search the database: ";
555
556 // Initiate endless while loop to request valid user input
557
558     while (true) {
559
560         // Capture user input
561
562         cin >> searchName;
563
564         // If input is invalid, restart loop
565
566         if (cin.fail()) {
567             cin.clear();
568             cin.ignore(1000, '\n');
569             cout << "The input you provided is not valid.
                ↳ Please try again: ";
570
571         // Iterate through the list of names and search for
                ↳ matching names
572
573         } else {
574             for (int i = 0; i < names.size(); i++) {
575                 if (names.at(i) == searchName && !nameFound) {
576
577                     // If name is found, set nameFound to true
578
579                     nameFound = true;
580
581                     // Set tempName and tempScore values and
                        ↳ send call to printList() function
582
583                     tempName.push_back(names.at(i));
584                     tempScore.push_back(scores.at(i));
585                     printList(tempName, tempScore, true);
586
587                 }
588             }
589
590             // If no matching name is found, inform the user

```



```

591
592         if (!nameFound) {
593             cout << "\nName not found." << endl;
594             waitForContinue();
595         }
596         // End loop
597         break;
598     }
599 }
600 }
601 }
602 }
603 }
604
605 // Search database for a score
606
607 void searchScore(vector<string> & names, vector<int> & scores) {
608     // Clear the console
609     clearConsole();
610     // Declare searchScore and scoreFound variables
611     int searchScore;
612     bool scoreFound = false;
613     // Declare tempName and tempScore variables to hold potential
614     ↪ name(s) and score(s)
615     vector<string> tempName;
616     vector<int> tempScore;
617     // Check that the database is not empty
618     if (checkDatabaseHasInputs(names)) {
619         // Request user input
620         cout << "Enter the score for which to search the database:
621         ↪ ";
622         // Initiate endless while loop to request valid user input
623         while (true) {
624             // Capture user input
625             cin >> searchScore;
626             // If input is invalid, restart loop
627             if (cin.fail()) {
628                 cin.clear();
629                 cin.ignore(1000, '\n');

```

```

644         cout << "The input you provided is not valid.
        ↳ Please try again: ";
645
646     // Iterate through the list of scores and capture all
        ↳ matching values
647     // For each matching value, push to the tempName and
        ↳ tempScore vectors
648
649     } else {
650         for (int i = 0; i < scores.size(); i++) {
651             if (scores.at(i) == searchScore) {
652                 scoreFound = true;
653                 tempName.push_back(names.at(i));
654                 tempScore.push_back(scores.at(i));
655
656                 // Send all matching names and scores to
                    ↳ printList() function for printing
657
658                 printList(tempName, tempScore, false);
659             }
660         }
661
662         // If no matching score is found, inform the user
663
664         if (!scoreFound) {
665             cout << "\nScore not found." << endl;
666         }
667
668         // Wait for user to indicate readiness
669
670         waitForContinue();
671
672         // End loop
673         break;
674     }
675 }
676 }
677 }
678 }

```
